全球有超过100 000的开发者使用本书来学习Spring
畅销经典 Spring 技术图书，针对 Spring 4 全新升级

Spring 实战

（第4版）

Spring IN ACTION
FOURTH EDITION

〔美〕Craig Walls 著
张卫滨 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

VISIT…

LANZAROTE Caliente.COM

目录

版权信息
作者简介
版权声明
内容提要
关于本书
    路线图
    代码规范与下载
    作者在线
    封面插图简介
前言
译者序
致谢
第1部分 Spring的核心
第1章 Spring之旅
    1.1 简化Java开发
        1.1.1 激发POJO的潜能
        1.1.2 依赖注入
        1.1.3 应用切面
        1.1.4 使用模板消除样板式代码
    1.2 容纳你的Bean
        1.2.1 使用应用上下文
        1.2.2 bean的生命周期
    1.3 俯瞰Spring风景线
        1.3.1 Spring模块
        1.3.2 Spring Portfolio
    1.4 Spring的新功能
        1.4.1 Spring 3.1新特性
        1.4.2 Spring 3.2新特性
        1.4.3 Spring 4.0新特性
    1.5 小结
第2章 装配Bean
    2.1 Spring配置的可选方案
    2.2 自动化装配bean
        2.2.1 创建可被发现的bean
        2.2.2 为组件扫描的bean命名
        2.2.3 设置组件扫描的基础包

2.2.4 通过为bean添加注解实现自动装配
2.2.5 验证自动装配
2.3 通过Java代码装配bean
2.3.1 创建配置类
2.3.2 声明简单的bean
2.3.3 借助JavaConfig实现注入
2.4 通过XML装配bean
2.4.1 创建XML配置规范
2.4.2 声明一个简单的<bean>
2.4.3 借助构造器注入初始化bean
2.4.4 设置属性
2.5 导入和混合配置
2.5.1 在JavaConfig中引用XML配置
2.5.2 在XML配置中引用JavaConfig
2.6 小结
第3章 高级装配
3.1 环境与profile
3.1.1 配置profile bean
3.1.2 激活profile
3.2 条件化的bean
3.3 处理自动装配的歧义性
3.3.1 标示首选的bean
3.3.2 限定自动装配的bean
3.4 bean的作用域
3.4.1 使用会话和请求作用域
3.4.2 在XML中声明作用域代理
3.5 运行时值注入
3.5.1 注入外部的值
3.5.2 使用Spring表达式语言进行装配
3.6 小结
第4章 面向切面的Spring
4.1 什么是面向切面编程
4.1.1 定义AOP术语
4.1.2 Spring对AOP的支持
4.2 通过切点来选择连接点
4.2.1 编写切点
4.2.2 在切点中选择bean
4.3 使用注解创建切面
4.3.1 定义切面
4.3.2 创建环绕通知

4.3.3 处理通知中的参数
4.3.4 通过注解引入新功能
4.4 在XML中声明切面
4.4.1 声明前置和后置通知
4.4.2 声明环绕通知
4.4.3 为通知传递参数
4.4.4 通过切面引入新的功能
4.5 注入AspectJ切面
4.6 小结
第2部分 Web中的Spring
第5章 构建Spring Web应用程序
5.1 Spring MVC起步
5.1.1 跟踪Spring MVC的请求
5.1.2 搭建Spring MVC
5.1.3 Spittr应用简介
5.2 编写基本的控制器
5.2.1 测试控制器
5.2.2 定义类级别的请求处理
5.2.3 传递模型数据到视图中
5.3 接受请求的输入
5.3.1 处理查询参数
5.3.2 通过路径参数接受输入
5.4 处理表单
5.4.1 编写处理表单的控制器
5.4.2 校验表单
5.5 小结
第6章 渲染Web视图
6.1 理解视图解析
6.2 创建JSP视图
6.2.1 配置适用于JSP的视图解析器
6.2.2 使用Spring的JSP库
6.3 使用Apache Tiles视图定义布局
6.3.1 配置Tiles视图解析器
6.4 使用Thymeleaf
6.4.1 配置Thymeleaf视图解析器
6.4.2 定义Thymeleaf模板
6.5 小结
第7章 Spring MVC的高级技术
7.1 Spring MVC配置的替代方案
7.1.1 自定义DispatcherServlet配置

7.1.2 添加其他的Servlet和Filter
7.1.3 在web.xml中声明DispatcherServlet
7.2 处理multipart形式的数据
7.2.1 配置multipart解析器
7.2.2 处理multipart请求
7.3 处理异常
7.3.1 将异常映射为HTTP状态码
7.3.2 编写异常处理的方法
7.4 为控制器添加通知
7.5 跨重定向请求传递数据
7.5.1 通过URL模板进行重定向
7.5.2 使用flash属性
7.6 小结
第8章 使用Spring Web Flow
8.1 在Spring中配置Web Flow
8.1.1 装配流程执行器
8.1.2 配置流程注册表
8.1.3 处理流程请求
8.2 流程的组件
8.2.1 状态
8.2.2 转移
8.2.3 流程数据
8.3 组合起来：披萨流程
8.3.1 定义基本流程
8.3.2 收集顾客信息
8.3.3 构建订单
8.3.4 支付
8.4 保护Web流程
8.5 小结
第9章 保护Web应用
9.1 Spring Security简介
9.1.1 理解Spring Security的模块
9.1.2 过滤Web请求
9.1.3 编写简单的安全性配置
9.2 选择查询用户详细信息的服务
9.2.1 使用基于内存的用户存储
9.2.2 基于数据库表进行认证
9.2.3 基于LDAP进行认证
9.2.4 配置自定义的用户服务
9.3 拦截请求

9.3.1 使用Spring表达式进行安全保护
9.3.2 强制通道的安全性
9.3.3 防止跨站请求伪造
9.4 认证用户
9.4.1 添加自定义的登录页
9.4.2 启用HTTP Basic认证
9.4.3 启用Remember-me功能
9.4.4 退出
9.5 保护视图
9.5.1 使用Spring Security的JSP标签库
9.5.2 使用Thymeleaf的Spring Security方言
9.6 小结
第3部分 后端中的Spring
第10章 通过Spring和JDBC征服数据库
10.1 Spring的数据访问哲学
10.1.1 了解Spring的数据访问异常体系
10.1.2 数据访问模板化
10.2 配置数据源
10.2.1 使用JNDI数据源
10.2.2 使用数据源连接池
10.2.3 基于JDBC驱动的数据源
10.2.4 使用嵌入式的数据源
10.2.5 使用profile选择数据源
10.3 在Spring中使用JDBC
10.3.1 应对失控的JDBC代码
10.3.2 使用JDBC模板
10.4 小结
第11章 使用对象-关系映射持久化数据
11.1 在Spring中集成Hibernate
11.1.1 声明Hibernate的Session工厂
11.1.2 构建不依赖于Spring的Hibernate代码
11.2 Spring与Java持久化API
11.2.1 配置实体管理器工厂
11.2.2 编写基于JPA的Repository
11.3 借助Spring Data实现自动化的JPA Repository
11.3.1 定义查询方法
11.3.2 声明自定义查询
11.3.3 混合自定义的功能
11.4 小结
第12章 使用NoSQL数据库

12.1 使用MongoDB持久化文档数据
12.1.1 启用MongoDB
12.1.2 为模型添加注解，实现MongoDB持久化
12.1.3 使用MongoTemplate访问MongoDB
12.1.4 编写MongoDB Repository
12.2 使用Neo4j操作图数据
12.2.1 配置Spring Data Neo4j
12.2.2 使用注解标注图实体
12.2.3 使用Neo4jTemplate
12.2.4 创建自动化的Neo4j Repository
12.3 使用Redis操作key-value数据
12.3.1 连接到Redis
12.3.2 使用RedisTemplate
12.3.3 使用key和value的序列化器
12.4 小结
第13章 缓存数据
13.1 启用对缓存的支持
13.1.1 配置缓存管理器
13.2 为方法添加注解以支持缓存
13.2.1 填充缓存
13.2.2 移除缓存条目
13.3 使用XML声明缓存
13.4 小结
第14章 保护方法应用
14.1 使用注解保护方法
14.1.1 使用@Secured注解限制方法调用
14.1.2 在Spring Security中使用JSR-250的@RolesAllowed注解
14.2 使用表达式实现方法级别的安全性
14.2.1 表述方法访问规则
14.2.2 过滤方法的输入和输出
14.3 小结
第4部分 Spring集成
第15章 使用远程服务
15.1 Spring远程调用概览
15.2 使用RMI
15.2.1 导出RMI服务
15.2.2 装配RMI服务
15.3 使用Hessian和Burlap发布远程服务
15.3.1 使用Hessian和Burlap导出bean的功能
15.3.2 访问Hessian/Burlap服务

15.4 使用Spring的HttpInvoker
15.4.1 将bean导出为HTTP服务
15.4.2 通过HTTP访问服务
15.5 发布和使用Web服务
15.5.1 创建基于Spring的JAX-WS端点
15.5.2 在客户端代理JAX-WS服务
15.6 小结
第16章 使用Spring MVC创建REST API
16.1 了解REST
16.1.1 REST的基础知识
16.1.2 Spring是如何支持REST的
16.2 创建第一个REST端点
16.2.1 协商资源表述
16.2.2 使用HTTP信息转换器
16.3 提供资源之外的其他内容
16.3.1 发送错误信息到客户端
16.3.2 在响应中设置头部信息
16.4 编写REST客户端
16.4.1 了解RestTemplate的操作
16.4.2 GET资源
16.4.3 检索资源
16.4.4 抽取响应的元数据
16.4.5 PUT资源
16.4.6 DELETE资源
16.4.7 POST资源数据
16.4.8 在POST请求中获取响应对象
16.4.9 在POST请求后获取资源位置
16.4.10 交换资源
16.5 小结
第17章 Spring消息
17.1 异步消息简介
17.1.1 发送消息
17.1.2 评估异步消息的优点
17.2 使用JMS发送消息
17.2.1 在Spring中搭建消息代理
17.2.2 使用Spring的JMS模板
17.2.3 创建消息驱动的POJO
17.2.4 使用基于消息的RPC
17.3 使用AMQP实现消息功能
17.3.1 AMQP简介

17.3.2 配置Spring支持AMQP消息
17.3.3 使用RabbitTemplate发送消息
17.3.4 接收AMQP消息
17.4 小结
第18章 使用WebSocket和STOMP实现消息功能
18.1 使用Spring的低层级WebSocket API
18.2 应对不支持WebSocket的场景
18.3 使用STOMP消息
18.3.1 启用STOMP消息功能
18.3.2 处理来自客户端的STOMP消息
18.3.3 发送消息到客户端
18.4 为目标用户发送消息
18.4.1 在控制器中处理用户的消息
18.4.2 为指定用户发送消息
18.5 处理消息异常
18.6 小结
第19章 使用Spring发送Email
19.1 配置Spring发送邮件
19.1.1 配置邮件发送器
19.1.2 装配和使用邮件发送器
19.2 构建丰富内容的Email消息
19.2.1 添加附件
19.2.2 发送富文本内容的Email
19.3 使用模板生成Email
19.3.1 使用Velocity构建Email消息
19.3.2 使用Thymeleaf构建Email消息
19.4 小结
第20章 使用JMX管理Spring Bean
20.1 将Spring bean导出为MBean
20.1.1 通过名称暴露方法
20.1.2 使用接口定义MBean的操作和属性
20.1.3 使用注解驱动的MBean
20.1.4 处理MBean冲突
20.2 远程MBean
20.2.1 暴露远程MBean
20.2.2 访问远程MBean
20.2.3 代理MBean
20.3 处理通知
20.3.1 监听通知
20.4 小结

第21章 借助Spring Boot简化Spring开发
21.1 Spring Boot简介
21.1.1 添加Starter依赖
21.1.2 自动配置
21.1.3 Spring Boot CLI
21.1.4 Actuator
21.2 使用Spring Boot构建应用
21.2.1 处理请求
21.2.2 创建视图
21.2.3 添加静态内容
21.2.4 持久化数据
21.2.5 尝试运行
21.3 组合使用Groovy与Spring Boot CLI
21.3.1 编写Groovy控制器
21.3.2 使用Groovy Repository实现数据持久化
21.3.3 运行Spring Boot CLI
21.4 通过Actuator获取了解应用内部状况
21.5 小结
看完了

版权信息

书名：Spring实战（第4版）

ISBN：978-7-115-41730-5

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

- 著　[美] Craig Walls

  译　张卫滨

  责任编辑　陈冀康

- 人民邮电出版社出版发行　北京市丰台区成寿寺路11号

  邮编 100164　电子邮件 315@ptpress.com.cn

  网址 http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

  反盗版热线：(010)81055315

作者简介

Craig Walls是Pivotal的高级工程师，是Spring Social和Spring Sync的项目领导者，同时也是Manning出版社《Spring In Action》的作者，目前这本书已经更新到了第四版。他非常热心于Spring框架的推广，经常在当地的用户组和会议上演讲并在博客上撰写Spring相关的内容。在不琢磨代码的时候，Craig Walls会尽可能多地陪伴他的妻子、两个女儿、两只小鸟以及两只小狗。

本书特色

全球有超过100 000的开发者使用本书来学习Spring

中文版累计销售超10万册，畅销经典Spring 技术图书，针对Spring 4全新升级

作者Craig Walls，SpringSource的软件开发人员，也是一位畅销书作者。

第3版译者继续翻译新版，品质保障！

版权声明

Original English language edition, entitled Spring in Action,4th Edition by Craig Walls Bibeault published by Manning Publications Co., 209 Bruce Park Avenue, Greenwich, CT 06830. Copyright ©2015 by Manning Publications Co.

Simplified Chinese-language edition copyright ©2016 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由**Manning Publications Co.**授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制本书内容。

版权所有，侵权必究。

内容提要

本书是经典的、畅销的Spring学习和实践指南。

第4版针对Spring 4进行了全面更新。全书分为4部分。第1部分介绍Spring框架的核心知识。第2部分在此基础上介绍了如何使用Spring构建Web应用程序。第3部分告别前端，介绍了如何在应用程序的后端使用Spring。第4部分描述了如何使用Spring与其他的应用和服务进行集成。

本书适用于已具有一定Java 编程基础的读者，以及在Java 平台下进行各类软件开发的开发人员、测试人员，尤其适用于企业级Java 开发人员。本书既可以被刚开始学习Spring 的读者当作学习指南，也可以被那些想深入了解Spring 某方面功能的资深用户作为参考用书。

关于本书

Spring框架是以简化Java EE应用程序的开发为目标而创建的。同样，本书是为了帮助读者更容易地使用Spring而编写的。我的目标不是为读者详细地列出Spring API，而是希望通过现实中的实际示例代码来为Java EE开发人员展现Spring框架。因为Spring是一个模块化的框架，所以这本书也是按照这种方式编写的。我们知道并不是所有的开发人员都有相同的需求，有些人想从头学习Spring，而有的可能只想排出几个主题，然后按照自己的节奏来学习。所以，本书既可以被刚开始学习Spring的读者当作学习指南，也可以被那些想深入了解某方面功能的读者作为参考。

本书适用于所有的Java开发人员，企业级Java开发人员将会发现更有帮助。我将会循序渐进地指导读者浏览本书中每章复杂的示例代码，但Spring的真正强大之处在于它能够使企业级应用程序的开发更简单。因此，企业级应用程序的开发人员会更加欣赏本书的示例代码。因为Spring的绝大部分内容都是提供企业级服务的，所以这里包含了许多Spring和EJB的比较。

路线图

本书分为4部分。第1部分介绍Spring框架的核心知识。第2部分在此基础上介绍如何使用Spring构建Web应用程序。第3部分告别前端，介绍如何在应用程序的后端使用Spring。第4部分描述如何使用Spring与其他的应用和服务进行集成。

在第1部分中，读者将会学习到Spring容器、依赖注入（dependency injection，DI）和面向切面编程（aspect-oriented programming，AOP），也就是Spring框架的核心。这能让读者很好地理解Spring的基础原理，而这些原理将会在本书各个章节都会用到。

- 第1章将会概要地介绍Spring，包括DI和AOP的一些基本样例。同时，读者还会了解到更大的Spring生态系统的整体情况。
- 第2章更为详细地介绍DI，展现应用程序中的各个组件（bean）如何装配在一起。这包括基于XML装配、基于Java装配以及自动装配。
- 在掌握了基本的bean装配后，第3章会介绍几种高级装配技术，

读者可能并不会经常用到这些技术，但是如果用到的话，本章的内容将会告诉读者如何发挥Spring容器最强大的威力。

- 第4章介绍如何使用Spring的AOP来为对象解耦那些对其提供服务的横切性关注点。这一章也为后面各章提供基础，在后面读者将会使用AOP来提供声明式服务，如事务、安全和缓存。

在第2部分中，读者将会看到如何使用Spring来构建Web应用程序。

- 第5章介绍使用Spring MVC的基础知识，这是Spring中的基础Web框架。读者将会看到如何编写控制器来处理请求，并使用模型数据产生响应。
- 当控制器的工作完成后，模型数据必须要使用一个视图来进行渲染。第6章将会探讨在Spring中可以使用的各种视图技术，包括JSP、Apache Tiles以及Thymeleaf。
- 第7章的内容不再是Spring MVC的基础知识了，在本章中，读者将会学习到如何自定义Spring MVC配置、处理multipart类型的文件上传、处理在控制器中可能会出现的异常并且会通过flash属性在请求之间传递数据。
- 第8章将会介绍Spring Web Flow，这是Spring MVC的一个扩展，能够开发会话式的Web应用程序。在本章中，读者将会学习到如何构建引导用户完成特定流程的Web应用程序。
- 第9章读者将会学到如何使用Spring Security为自己的应用程序Web层实现安全性。

第3部分所关注的内容不再是应用程序的前端了，而是关注于如何处理和持久化数据。

- 第10章首先会介绍如何使用Spring对JDBC的抽象实现关系型数据库中的数据持久化。
- 第11章从另外一个角度介绍数据持久化，也就是使用Java持久化API（JPA）存储关系型数据库中的数据。
- 第12章将会介绍如何将Spring与非关系型数据库结合使用，如MongoDB和Neo4j。
- 不管数据存储在什么地方，缓存都有助于性能的提升，这是通过只有在必要的时候才去查询数据库实现的。第13章将会为读者介绍Spring对声明式缓存的支持。
- 第14章重新回到Spring Security，将会介绍如何通过AOP将安全性应用到方法级别。

本书的最后一部分会介绍如何将Spring应用程序与其他系统进行集成。

- 第15章将会学习如何创建与使用远程服务，包括RMI、Hessian、Burlap以及基于SOAP的服务。
- 第16章将会再次回到Spring MVC，我们将会看到如何创建RESTful服务，在这个过程中所使用的编程模型与之前在第5章中所描述的是一致的。
- 第17章将会探讨Spring对异步消息的支持，本章将会包括Java消息服务（Java Message Service，JMS）以及高级消息队列协议（Advanced Message Queuing Protocol，AMQP）。
- 在第18章中，异步消息有了新的花样，在这一章中读者会看到如何将Spring与WebSocket和STOMP结合起来，实现服务端与客户端之间的异步通信。
- 第19章将会介绍如何使用Spring发送E-mail。
- 第20章会关注于Spring对Java管理扩展（Java Management Extensions，JMX）功能的支持，借助这项功能可以对Spring应用程序进行监控和修改运行时配置。
- 最后，在第21章，读者将会看到一个全新并且会改变游戏规则的Spring使用方式，名为Spring Boot。我们将会看到Spring Boot如何将Spring应用中样板式的配置移除掉，这样就能让读者更加专注于业务功能。

代码规范与下载

本书中有大量的示例代码。这些代码将会使用固定宽度的代码字体。本书正文中的类名、方法名或XML片段也都使用代码字体。

很多Spring类和包的名字很长（不过会有较强的表达性）。鉴于此，我们有时候会用到换行符（➥）。

本书中的示例代码并不都是完整的。为了关注某个主题，我有时候只会展示类的一个或两个方法。本书所构建的应用程序完整代码可以在出版社站点上下载，地址是www.manning.com/SpringinActionFourthEdition。

作者在线

购买了本书，读者就可以免费访问Manning出版社提供的在线论坛，在这里读者可以给本书写评论，问一些技术问题并可以得到作者和其他用户的帮助。要进入这个论坛或订阅它，读者可以在浏览器中访问www.manning.com/SpringinActionFourthEdition。这个页面会告诉读者

注册后怎样进入论坛，能够得到什么帮助以及论坛的规则。

Manning对读者的许诺是为读者提供一个交流平台，在这里读者之间以及读者和作者之间可以进行有意义的交流。对于作者来说，对论坛进行多少次的访问不是强制的，他们对本书论坛的贡献是自愿和免费的。我们建议读者尽量向作者问一些有挑战性的问题，以保持他们的兴趣！

只要本书还在发售，读者就可以访问作者在线论坛以及以前讨论的归档信息。

封面插图简介

《Spring实战》第4版的封面人物是“Le Caraco”，也就是约旦西南部卡拉克（Karak）省的居民。该省的首府是Al-Karak，那里的山顶有座古城堡，对死海和周边的平原有着极佳的视野。这幅图出自1796年出版的法国旅游图书，*Encyclopédiedes Voyages*，该书由J．G．St. Sauveur编写。在那时，为了娱乐而去旅游还是相对新鲜的做法，而像这样的旅游指南是很流行的，它能够让旅行家和足不出户的人们了解法国其他地区和国外的居民。

*Encyclopédiedes Voyages*中多种多样的图画生动描绘了200年前世界上各个城镇和地区的独特魅力。在那时，相隔数十千米的两个地区着装就不相同，可以通过着装判断人们究竟属于哪个地区。这本旅行指南展现了那个时代和其他历史时代的隔离感和距离感，这与我们这个运动过度的时代是截然不同的。

从那以后，服装风格发生了改变，富有地方特色的多样性开始淡化。现在，有时很难说一个洲的居民和其他洲的居民有什么不同。从积极的方面来看，我们或许是用原来文化和视觉上的多样性换来了个人风格的多变性，或者可以说是更为多样化和有趣的知识科技生活。

这本旅行指南中的图片反映了两个世纪前各个地区生活的多样性，我们现在用图书封面的方式对其进行了再现。Manning出版社的员工都认为这是计算机行业中一个很有意思的创意。

前言

百尺竿头更进一步。十几年前，Spring刚刚进入Java开发领域，其目标是简化企业级Java开发。它使用更为简单和轻量级的模型，该模型基于简单老式的Java对象，以此挑战了当时重量级的开发模型。

现在，已经过去了很多年，Spring也发布了众多的版本，我们可以看到Spring在企业级应用开发领域已经有了巨大的影响力。对于无数的Java项目来说，它就是事实上的标准，并且对于一些规范和它本来想取代的框架，Spring也对其演进产生了影响。毫无疑问，如果Spring不挑战之前版本的企业级JavaBean（EJB）规范的话，现在的EJB规范肯定是完全不同的一个样子。

但是，Spring本身也在持续地演化和提升，它一直致力于将困难的开发任务进行简化，不断地为Java开发人员带来创新性的特性。在Spring最初所挑战的领域，Spring已经突飞猛进，涉及的范围扩展到Java应用开发的各个方面。

因此，为了介绍Spring的现状，我们需要对这本书升级了。在本书上一版出版到现在的几年间，发生了太多的事情，想在这一版中将所有的变化都涵盖进来是不可能的。不过，在第4版的《Spring实战》中，我依然会使其包含尽可能多的内容。下面列出了在这一版中新增的一些令人兴奋的新内容：

- 强调基于Java的Spring配置，基于Java的配置方案几乎可以用在所有Spring开发领域之中；
- 条件化的配置以及profile特性能够让Spring在运行时确定该使用或忽略哪些Spring配置；
- Spring MVC的多项增强和改善，尤其是与创建REST服务相关的；
- 在Spring应用中使用Thymeleaf替代JSP；
- 使用基于Java的配置启用Spring Security；
- 使用Spring Data，在运行时自动为JPA、MongoDB和Neo4j生成Repository实现；
- Spring新提供的声明式缓存支持；
- 借助WebSocket和STOMP，实现异步的Web消息；
- Spring Boot，改变使用Spring游戏规则的新方法。

如果在Spring方面读者已经有相当多经验的话，那么将会发现这些新元素对于自己的Spring工具箱来说是非常有价值的补充。如果读者是要学习Spring的新手，那么就赶上了学习Spring的一个好时代，这本书会帮助读者起步。

对于Spring的使用来说，这的确是一个令人兴奋的时代。在过去的12年里，在使用Spring进行开发以及编写与之相关的文章方面形成了一股浪潮。我迫不及待地想看到Spring接下来会做些什么！

译者序

3年前，有幸和耿渊同学合作翻译了《Spring实战（第3版）》。3年的时光过去了，技术在不断发展，这本书也推出了最新的第4版，顺利将这本书翻译完成后，顿时感觉轻松了许多。译书是一件比较辛苦的工作，但是在这3年的时间内，每当看到有朋友选择本书来学习Spring，自己觉得还是蛮有成就感的。所以，看到本书的第4版时，我迫不及待地联系编辑约定了本书的翻译事宜。

本书的作者Craig Walls先生，从10年前编写本书的第1版开始，持续把一件事情做好，紧跟技术的发展，不断地升级和更新这本书的内容，世界范围内无数的Java开发者通过这本书学习和掌握了Spring技术。

本书的主题是Spring框架，从十多年前问世以来，它一直致力于简化JEE应用的开发。从最初的挑战者，到现在诸多标准的制定者；从传统的JEE应用，到大数据、NoSQL、企业应用集成、批处理、移动开发等领域，Spring都在参与和发挥影响力。新版本的Spring提供了更加丰富的功能，但更重要的是Spring在想尽办法简化开发人员的使用，包括自动配置、基于Java的配置，还有现在越来越受到欢迎的Spring Boot。Spring Boot是对Spring本身的一种颠覆和革命，但是唯有这种颠覆，才会换来开发人员更多的喜爱和框架本身的发展。

这本书从第1版到第4版之所以长盛不衰，是因为它紧跟了技术的发展；Spring十多年来一直受到Java开发者的青睐，是因为它不断地进步和改善，并且坚持最初的目标：简化企业级Java的开发。处于一个不断革新的领域，我们技术人员何尝不需要如此呢，只有不断地汲取新的知识，学习新的技术，才能保证不被时代所淘汰。

本书涵盖了Spring框架的许多领域，既有核心框架，也有各种功能扩展，不少的同学曾经对我言及，感觉书中所讲述的内容深度不够，但是我个人认为，对于开源框架的学习，我们会有不同的掌握深度，从最初始的使用、配置，到设计原理，再到源码分析，一本书很难面面俱到深入介绍所有的内容，但是它却能够提供一个方向，让我们按图索骥深入学习更多的知识。

译书占用了大量的业余时间，因此感谢我的爱人，帮我承担了许多家务和带孩子的工作，还要感谢我的儿子，每天看到他的成长和进步，

都让我感觉如果懈怠的话，该被小朋友嘲笑了。

尽管在翻译的过程中，我力争达到准确和通畅，并与作者进行了很多的沟通和交流，但限于水平和时间，肯定还有许多的不足或纰漏之处，热忱期待您提出意见，希望本书能够对您有用！您可以通过levinzhang1981@126.com联系到我。

张卫滨

2015年11月于大连

致谢

在本书付印之前，在本书捆扎之前，在本书装箱之前，在本书交付运输之前，在本书到达你手里之前，在整个过程中，有很多双手都曾经接触过它。即便你阅读电子版，省去了上面所述的流程，在你所下载的位和字节上依然凝结着很多双手的辛勤劳动——编辑、审阅、录入以及校对。如果没有这么多人的付出，这本书也就不会存在了。

首先，我要感谢Manning辛苦工作的每个人，当这本书的进展速度没有达到预期时，他们给予了足够的耐心，并促使我完成这本书：Marjan Bace、Michael Stephens、Cynthia Kane、Andy Carroll、Benjamin Berg、Alyson Brener、Dottie Marisco、Mary Piergies、Janet Vail以及幕后的其他很多人。

写书的时候，尽早和频繁的反馈是相当重要的，这一点与开发软件是一样的。当这本书还非常粗糙的时候，有些人审阅了初稿并提供反馈，帮助本书最终成型。要感谢下面的人：Bob Casazza、Chaoho Hsieh、Christophe Martini、Gregor Zurowski、James Wright、Jeelani Basha、Jens Richter、Jonathan Thoms、Josh Hart、Karen Christenson、Mario Arias、Michael Roberts、Paul Balogh、Ricardo da Silva Lima。尤其要感谢John Ryan，在本书交付前，他对书稿进行了全面的技术审校。

当然，我要感谢美丽的妻子，感谢她容忍我开始了这个新的写作工程，感谢她整个过程中所给予我的鼓励。我深深地爱着你。

Maisy和Madi，世界上最可爱的小姑娘，感谢你们的拥抱、欢笑以及对本书内容别出心裁的见解。

对于Spring团队的同事，怎么说呢？你们太酷了！能够作为推动Spring前进的团队中的一员，我感到非常荣幸和感激。你们层出不穷的新创意总是让我感到惊叹。

感谢我在用户组和No Fluff/Just Stuff会议上演讲时所遇到的每个人。

最后，感谢Phoenicians，你们（以及Epcot）太棒了！[1]

---

[1] Phoenicians指的是远古时代的腓尼基人，他们被认为是字母系统的创建者，基于字母的所有现代语言都由此衍生而来。在迪斯尼世界的Epcot，有名为Spaceship Earth的时光穿梭体验，我们可以了解到人类交流的历史，甚至能够回到腓尼基人的时代，在这段旅程的旁白中这样说道：如果你觉得学习字母语言很容易的话，那感谢腓尼基人吧，是他们发明了它。这是作者的一种幽默说法。——译者注

第1部分　Spring的核心

Spring可以做很多事情，它为企业级开发提供给了丰富的功能，但是这些功能的底层都依赖于它的两个核心特性，也就是依赖注入（dependency injection，DI）和面向切面编程（aspect-oriented programming，AOP）。

作为本书的开始，在第1章“Spring之旅”中，我将快速介绍一下Spring框架，包括Spring DI和AOP的概况，以及它们是如何帮助读者解耦应用组件的。

在第2章“装配Bean”中，我们将深入探讨如何将应用中的各个组件拼装在一起，读者将会看到Spring所提供的自动配置、基于Java的配置以及XML配置。

在第3章“高级装配”中，将会告别基础的内容，为读者展现一些最大化Spring威力的技巧和技术，包括条件化装配、处理自动装配时的歧义性、作用域以及Spring表达式语言。

在第4章“面向切面的Spring”中，展示如何使用Spring的AOP特性把系统级的服务（例如安全和审计）从它们所服务的对象中解耦出来。本章也为后面的第9章、第13章和第14章做了铺垫，这几章将会分别介绍如何将Spring AOP用于声明式安全以及缓存。

第1章　Spring之旅

本章内容：

- Spring的bean容器
- 介绍Spring的核心模块
- 更为强大的Spring生态系统
- Spring的新功能

对于Java程序员来说，这是一个很好的时代。

在Java近20年的历史中，它经历过很好的时代，也经历过饱受诟病的时代。尽管有很多粗糙的地方，如applet、企业级JavaBean（Enterprise JavaBean，EJB）、Java数据对象（Java Data Object，JDO）以及无数的日志框架，但是作为一个平台，Java的历史是丰富多彩的，有很多的企业级软件都是基于这个平台构建的。Spring是Java历史中很重要的组成部分。

在诞生之初，创建Spring的主要目的是用来替代更加重量级的企业级Java技术，尤其是EJB。相对于EJB来说，Spring提供了更加轻量级和简单的编程模型。它增强了简单老式Java对象（Plain Old Java object，POJO）的功能，使其具备了之前只有EJB和其他企业级Java规范才具有的功能。

随着时间的推移，EJB以及Java 2企业版（Java 2 Enterprise Edition，J2EE）在不断演化。EJB自身也提供了面向简单POJO的编程模型。现在，EJB也采用了依赖注入（Dependency Injection，DI）和面向切面编程（Aspect-Oriented Programming，AOP）的理念，这毫无疑问是受到Spring成功的启发。

尽管J2EE（现在称之为JEE）能够赶上Spring的步伐，但Spring也没有停止前进。Spring继续在其他领域发展，而JEE则刚刚开始涉及这些领域，或者还完全没有开始在这些领域的创新。移动开发、社交API集成、NoSQL数据库、云计算以及大数据都是Spring正在涉足和创新的领域。Spring的前景依然会很美好。

正如我之前所言，对于Java开发者来说，这是一个很好的时代。

本书会对Spring进行研究，在这一章中，我们将会在较为宏观的层面

上介绍Spring，让你对Spring是什么有直观的体验。本章将让读者对Spring所解决的各类问题有一个清晰的认识，同时为其他章奠定基础。

1.1 简化Java开发

Spring是一个开源框架，最早由Rod Johnson创建，并在《Expert One-on-One：J2EE Design and Development》（http://amzn.com/076454385）这本著作中进行了介绍。Spring是为了解决企业级应用开发的复杂性而创建的，使用Spring可以让简单的JavaBean实现之前只有EJB才能完成的事情。但Spring不仅仅局限于服务器端开发，任何Java应用都能在简单性、可测试性和松耦合等方面从Spring中获益。

bean的各种名称……虽然Spring用bean或者JavaBean来表示应用组件，但并不意味着Spring组件必须要遵循JavaBean规范。一个Spring组件可以是任何形式的POJO。在本书中，我采用JavaBean的广泛定义，即POJO的同义词。

纵览全书，读者会发现Spring 可以做非常多的事情。但归根结底，支撑Spring的仅仅是少许的基本理念，所有的理念都可以追溯到Spring最根本的使命上：简化Java开发。

这是一个郑重的承诺。许多框架都声称在某些方面做了简化，但Spring的目标是致力于全方位的简化Java开发。这势必引出更多的解释，Spring是如何简化Java开发的？

为了降低Java开发的复杂性，Spring采取了以下4种关键策略：

- 基于POJO的轻量级和最小侵入性编程；
- 通过依赖注入和面向接口实现松耦合；
- 基于切面和惯例进行声明式编程；
- 通过切面和模板减少样板式代码。

几乎Spring所做的任何事情都可以追溯到上述的一条或多条策略。在本章的其他部分，我将通过具体的案例进一步阐述这些理念，以此来证明Spring是如何完美兑现它的承诺的，也就是简化Java开发。让我们先从基于POJO的最小侵入性编程开始。

1.1.1 激发POJO的潜能

如果你从事Java编程有一段时间了，那么你或许会发现（可能你也实际使用过）很多框架通过强迫应用继承它们的类或实现它们的接口从而导致应用与框架绑死。一个典型的例子是EJB 2时代的无状态会话bean。早期的EJB是一个很容易想到的例子，不过这种侵入式的编程方式在早期版本的Struts、WebWork、Tapestry以及无数其他的Java规范和框架中都能看到。

Spring竭力避免因自身的API而弄乱你的应用代码。Spring不会强迫你实现Spring规范的接口或继承Spring规范的类，相反，在基于Spring构建的应用中，它的类通常没有任何痕迹表明你使用了Spring。最坏的场景是，一个类或许会使用Spring注解，但它依旧是POJO。

不妨举个例子，请参考下面的`HelloWorldBean`类：

程序清单**1.1　Spring**不会在**HelloWorldBean**上有任何不合理的要求

```
package com.habuma.spring;
public class HelloWorldBean {
  public String sayHello() {          <— 这就是你所需要做的
    return "Hello World";
  }
}
```

可以看到，这是一个简单普通的Java类——POJO。没有任何地方表明它是一个Spring组件。Spring的非侵入编程模型意味着这个类在Spring应用和非Spring应用中都可以发挥同样的作用。

尽管形式看起来很简单，但POJO一样可以具有魔力。Spring赋予POJO魔力的方式之一就是通过DI来装配它们。让我们看看DI是如何帮助应用对象彼此之间保持松散耦合的。

1.1.2　依赖注入

依赖注入这个词让人望而生畏，现在已经演变成一项复杂的编程技巧或设计模式理念。但事实证明，依赖注入并不像它听上去那么复杂。在项目中应用DI，你会发现你的代码会变得异常简单并且更容易理解和测试。

DI功能是如何实现的

任何一个有实际意义的应用（肯定比Hello World示例更复杂）都会由两个或者更多的类组成，这些类相互之间进行协作来完成特定的业务逻辑。按照传统的做法，每个对象负责管理与自己相互协作的对象

（即它所依赖的对象）的引用，这将会导致高度耦合和难以测试的代码。

举个例子，考虑下程序清单1.2所展现的Knight类。

程序清单**1.2　DamselRescuingKnight**只能执行**RescueDamselQuest**探险任务

```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight {

  private RescueDamselQuest quest;

  public DamselRescuingKnight() {
    this.quest = new RescueDamselQuest();
  }

  public void embarkOnQuest() {
    quest.embark();
  }

}
```

与 RescueDamselQuest 紧耦合

可以看到，DamselRescuingKnight在它的构造函数中自行创建了Rescue DamselQuest。这使得DamselRescuingKnight紧密地和RescueDamselQuest耦合到了一起，因此极大地限制了这个骑士执行探险的能力。如果一个少女需要救援，这个骑士能够召之即来。但是如果一条恶龙需要杀掉，或者一个圆桌……额……需要滚起来，那么这个骑士就爱莫能助了。

更糟糕的是，为这个DamselRescuingKnight编写单元测试将出奇地困难。在这样的一个测试中，你必须保证当骑士的embarkOnQuest()方法被调用的时候，探险的embark()方法也要被调用。但是没有一个简单明了的方式能够实现这一点。很遗憾，DamselRescuingKnight将无法进行测试。

耦合具有两面性（two-headed beast）。一方面，紧密耦合的代码难以测试、难以复用、难以理解，并且典型地表现出“打地鼠”式的bug特性（修复一个bug，将会出现一个或者更多新的bug）。另一方面，一定程度的耦合又是必须的——完全没有耦合的代码什么也做不了。为了完成有实际意义的功能，不同的类必须以适当的方式进行交互。总而言之，耦合是必须的，但应当被小心谨慎地管理。

通过DI，对象的依赖关系将由系统中负责协调各对象的第三方组件在创建对象的时候进行设定。对象无需自行创建或管理它们的依赖关

系，如图1.1所示，依赖关系将被自动注入到需要它们的对象当中去。

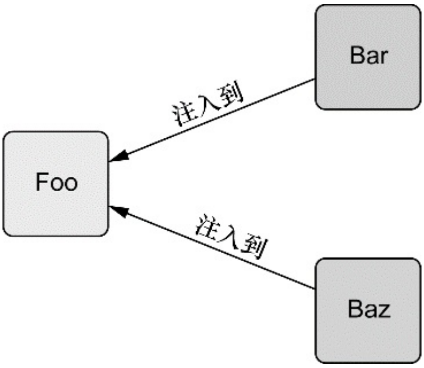

图1.1　依赖注入会将所依赖的关系自动交给目标对象，而不是让对象自己去获取依赖

为了展示这一点，让我们看一看程序清单1.3中的BraveKnight，这个骑士不仅勇敢，而且能挑战任何形式的探险。

程序清单**1.3　BraveKnight**足够灵活可以接受任何赋予他的探险任务

```
package com.springinaction.knights;

public class BraveKnight implements Knight {

  private Quest quest;

  public BraveKnight(Quest quest) {      <—— Quest 被注入进来
    this.quest = quest;
  }

  public void embarkOnQuest() {
    quest.embark();
  }

}
```

我们可以看到，不同于之前的DamselRescuingKnight，BraveKnight没有自行创建探险任务，而是在构造的时候把探险任务作为构造器参数传入。这是依赖注入的方式之一，即构造器注入（constructor injection）。

更重要的是，传入的探险类型是Quest，也就是所有探险任务都必须实现的一个接口。所以，BraveKnight能够响应RescueDamselQuest、SlayDragonQuest、MakeRoundTableRounderQuest等任意的Quest实现。

这里的要点是BraveKnight没有与任何特定的Quest实现发生耦合。对它来说，被要求挑战的探险任务只要实现了Quest接口，那么具体是哪种类型的探险就无关紧要了。这就是DI所带来的最大收益——松耦合。如果一个对象只通过接口（而不是具体实现或初始化过程）来表明依赖关系，那么这种依赖就能够在对象本身毫不知情的情况下，用不同的具体实现进行替换。

对依赖进行替换的一个最常用方法就是在测试的时候使用mock实现。我们无法充分地测试DamselRescuingKnight，因为它是紧耦合的；但是可以轻松地测试BraveKnight，只需给它一个Quest的mock实现即可，如程序清单1.4所示。

程序清单**1.4**　为了测试**BraveKnight**，需要注入一个**mock Quest**

```
package com.springinaction.knights;
import static org.mockito.Mockito.*;
import org.junit.Test;

public class BraveKnightTest {

  @Test
  public void knightShouldEmbarkOnQuest() {
    Quest mockQuest = mock(Quest.class);                    <-- 创建 mock Quest
    BraveKnight knight = new BraveKnight(mockQuest);      <-- 注入 mock Quest
    knight.embarkOnQuest();
    verify(mockQuest, times(1)).embark();
  }

}
```

你可以使用mock框架Mockito去创建一个Quest接口的mock实现。通过这个mock对象，就可以创建一个新的BraveKnight实例，并通过构造器注入这个mock Quest。当调用embarkOnQuest()方法时，你可以要求Mockito框架验证Quest的mock实现的embark()方法仅仅被调用了一次。

将**Quest**注入到**Knight**中

现在BraveKnight类可以接受你传递给它的任意一种Quest的实现，但该怎样把特定的Query实现传给它呢？假设，希望BraveKnight所要进行探险任务是杀死一只怪龙，那么程序清单1.5中的SlayDragonQuest也许是挺合适的。

程序清单**1.5**　**SlayDragonQuest**是要注入到**BraveKnight**中的**Quest**实现

```
package com.springinaction.knights;

import java.io.PrintStream;

public class SlayDragonQuest implements Quest {

  private PrintStream stream;

  public SlayDragonQuest(PrintStream stream) {
    this.stream = stream;
  }

  public void embark() {
    stream.println("Embarking on quest to slay the dragon!");
  }

}
```

我们可以看到，SlayDragonQuest实现了Quest接口，这样它就适合注入到BraveKnight中去了。与其他的Java入门样例有所不同，SlayDragonQuest没有使用System.out.println()，而是在构造方法中请求一个更为通用的PrintStream。这里最大的问题在于，我们该如何将SlayDragonQuest交给BraveKnight呢？又如何将PrintStream交给SlayDragonQuest呢？

创建应用组件之间协作的行为通常称为装配（wiring）。Spring有多种装配bean的方式，采用XML是很常见的一种装配方式。程序清单1.6展现了一个简单的Spring配置文件：knights.xml，该配置文件将BraveKnight、SlayDragonQuest和PrintStream装配到了一起。

程序清单**1.6**　使用**Spring**将**SlayDragonQuest**注入到**BraveKnight**中

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="knight" class="com.springinaction.knights.BraveKnight">
    <constructor-arg ref="quest" />          <-- 注入 Quest bean
  </bean>

  <bean id="quest" class="com.springinaction.knights.SlayDragonQuest">   <-- 创建 SlayDragonQuest
    <constructor-arg value="#{T(System).out}" />
  </bean>

</beans>
```

在这里，BraveKnight和SlayDragonQuest被声明为Spring中的bean。就BraveKnight bean来讲，它在构造时传入了对

SlayDragonQuest bean的引用，将其作为构造器参数。同时，SlayDragonQuest bean的声明使用了Spring表达式语言（Spring Expression Language），将System.out（这是一个PrintStream）传入到了SlayDragonQuest的构造器中。

如果XML配置不符合你的喜好的话，Spring还支持使用Java来描述配置。比如，程序清单1.7展现了基于Java的配置，它的功能与程序清单1.6相同。

程序清单**1.7　Spring**提供了基于**Java**的配置，可作为**XML**的替代方案

```
package com.springinaction.knights.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.springinaction.knights.BraveKnight;
import com.springinaction.knights.Knight;
import com.springinaction.knights.Quest;
import com.springinaction.knights.SlayDragonQuest;

@Configuration
public class KnightConfig {

  @Bean
  public Knight knight() {
    return new BraveKnight(quest());
  }

  @Bean
  public Quest quest() {
    return new SlayDragonQuest(System.out);
  }

}
```

不管你使用的是基于XML的配置还是基于Java的配置，DI所带来的收益都是相同的。尽管BraveKnight依赖于Quest，但是它并不知道传递给它的是什么类型的Quest，也不知道这个Quest来自哪里。与之类似，SlayDragonQuest依赖于PrintStream，但是在编码时它并不需要知道这个PrintStream是什么样子的。只有Spring通过它的配置，能够了解这些组成部分是如何装配起来的。这样的话，就可以在不改变所依赖的类的情况下，修改依赖关系。

这个样例展现了在Spring中装配bean的一种简单方法。谨记现在不要过多关注细节。第2章我们会深入讲解Spring的配置文件，同时还会了解Spring装配bean的其他方式，甚至包括一种让Spring自动发现bean并在这些bean之间建立关联关系的方式。

现在已经声明了BraveKnight和Quest的关系，接下来我们只需要装载XML配置文件，并把应用启动起来。

观察它如何工作

Spring通过应用上下文（Application Context）装载bean的定义并把它们组装起来。Spring应用上下文全权负责对象的创建和组装。Spring自带了多种应用上下文的实现，它们之间主要的区别仅仅在于如何加载配置。

因为knights.xml中的bean是使用XML文件进行配置的，所以选择ClassPathXmlApplicationContext[1]作为应用上下文相对是比较合适的。该类加载位于应用程序类路径下的一个或多个XML配置文件。程序清单1.8中的main()方法调用ClassPathXmlApplicationContext加载knights.xml，并获得Knight对象的引用。

程序清单**1.8　KnightMain.java**加载包含**Knight**的**Spring**上下文

```
package com.springinaction.knights;

import org.springframework.context.support.
                  ClassPathXmlApplicationContext;

public class KnightMain {

  public static void main(String[] args) throws Exception {
    ClassPathXmlApplicationContext context =
        new ClassPathXmlApplicationContext(
            "META-INF/spring/knights.xml");
    Knight knight = context.getBean(Knight.class);
    knight.embarkOnQuest();
     context.close();
  }

}
```

加载 Spring 上下文

获取 knight bean

使用 knight

这里的main()方法基于knights.xml文件创建了Spring应用上下文。随后它调用该应用上下文获取一个ID为knight的bean。得到Knight对象的引用后，只需简单调用embarkOnQuest()方法就可以执行所赋予的探险任务了。注意这个类完全不知道我们的英雄骑士接受哪种探险任务，而且完全没有意识到这是由BraveKnight来执行的。只有knights.xml文件知道哪个骑士执行哪种探险任务。

通过示例我们对依赖注入进行了一个快速介绍。纵览全书，你将对依赖注入有更多的认识。如果你想了解更多关于依赖注入的信息，我推荐阅读Dhanji R. Prasanna的《Dependency Injection》，该著作覆盖了

依赖注入的所有内容。

现在让我们再关注Spring简化Java开发的下一个理念：基于切面进行声明式编程。

1.1.3 应用切面

DI能够让相互协作的软件组件保持松散耦合，而面向切面编程（aspect-oriented programming，AOP）允许你把遍布应用各处的功能分离出来形成可重用的组件。

面向切面编程往往被定义为促使软件系统实现关注点的分离一项技术。系统由许多不同的组件组成，每一个组件各负责一块特定功能。除了实现自身核心的功能之外，这些组件还经常承担着额外的职责。诸如日志、事务管理和安全这样的系统服务经常融入到自身具有核心业务逻辑的组件中去，这些系统服务通常被称为横切关注点，因为它们会跨越系统的多个组件。

如果将这些关注点分散到多个组件中去，你的代码将会带来双重的复杂性。

- 实现系统关注点功能的代码将会重复出现在多个组件中。这意味着如果你要改变这些关注点的逻辑，必须修改各个模块中的相关实现。即使你把这些关注点抽象为一个独立的模块，其他模块只是调用它的方法，但方法的调用还是会重复出现在各个模块中。
- 组件会因为那些与自身核心业务无关的代码而变得混乱。一个向地址簿增加地址条目的方法应该只关注如何添加地址，而不应该关注它是不是安全的或者是否需要支持事务。

图1.2展示了这种复杂性。左边的业务对象与系统级服务结合得过于紧密。每个对象不但要知道它需要记日志、进行安全控制和参与事务，还要亲自执行这些服务。

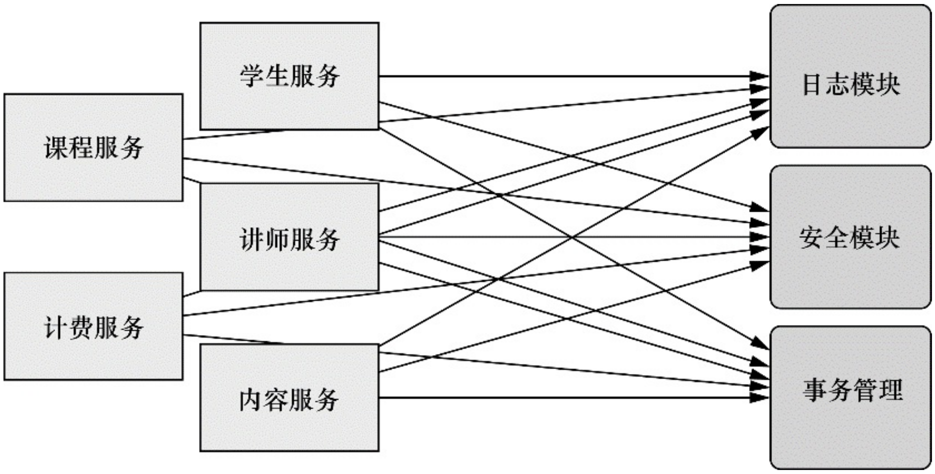

图1.2　在整个系统内，关注点（例如日志和安全）的调用经常散布到各个模块中，而这些关注点并不是模块的核心业务

AOP能够使这些服务模块化，并以声明的方式将它们应用到它们需要影响的组件中去。所造成的结果就是这些组件会具有更高的内聚性并且会更加关注自身的业务，完全不需要了解涉及系统服务所带来复杂性。总之，AOP能够确保POJO的简单性。

如图1.3所示，我们可以把切面想象为覆盖在很多组件之上的一个外壳。应用是由那些实现各自业务功能的模块组成的。借助AOP，可以使用各种功能层去包裹核心业务层。这些层以声明的方式灵活地应用到系统中，你的核心应用甚至根本不知道它们的存在。这是一个非常强大的理念，可以将安全、事务和日志关注点与核心业务逻辑相分离。

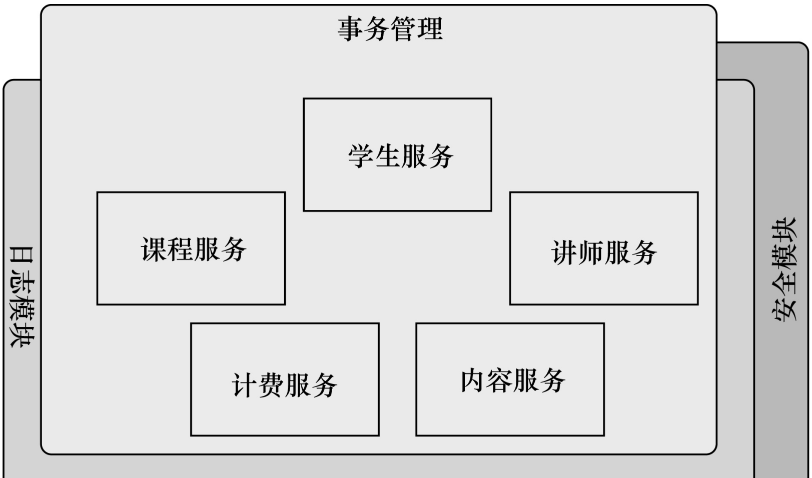

图1.3　利用AOP，系统范围内的关注点覆盖在它们所影响组件之上

为了示范在Spring中如何应用切面，让我们重新回到骑士的例子，并

为它添加一个切面。

AOP应用

每一个人都熟知骑士所做的任何事情，这是因为吟游诗人用诗歌记载了骑士的事迹并将其进行传唱。假设我们需要使用吟游诗人这个服务类来记载骑士的所有事迹。程序清单1.9展示了我们会使用的Minstrel类。

程序清单**1.9**　吟游诗人是中世纪的音乐记录器

```
package com.springinaction.knights;

import java.io.PrintStream;

public class Minstrel {

  private PrintStream stream;

  public Minstrel(PrintStream stream) {
    this.stream = stream;
  }

  public void singBeforeQuest() {                    <— 探险之前调用
    stream.println("Fa la la, the knight is so brave!");
  }

  public void singAfterQuest() {                      <— 探险之后调用
    stream.println("Tee hee hee, the brave knight " +
        "did embark on a quest!");
  }

}
```

正如你所看到的那样，Minstrel是只有两个方法的简单类。在骑士执行每一个探险任务之前，singBeforeQuest()方法会被调用；在骑士完成探险任务之后，singAfterQuest()方法会被调用。在这两种情况下，Minstrel都会通过一个PrintStream类来歌颂骑士的事迹，这个类是通过构造器注入进来的。

把Minstrel加入你的代码中并使其运行起来，这对你来说是小事一桩。我们适当做一下调整从而让BraveKnight可以使用Minstrel。程序清单1.10展示了将BraveKnight和Minstrel组合起来的第一次尝试。

程序清单**1.10**　**BraveKnight**必须要调用**Minstrel**的方法

```java
package com.springinaction.knights;

public class BraveKnight implements Knight {


  private Quest quest;
  private Minstrel minstrel;

  public BraveKnight(Quest quest, Minstrel minstrel) {
    this.quest = quest;
    this.minstrel = minstrel;
  }

  public void embarkOnQuest() throws QuestException {
    minstrel.singBeforeQuest();
    quest.embark();
    minstrel.singAfterQuest();
  }

}
```

Knight 应该管理它的 Minstrel 吗?

这应该可以达到预期效果。现在，你所需要做的就是回到Spring配置中，声明Minstrel bean并将其注入到BraveKnight的构造器之中。但是，请稍等……

我们似乎感觉有些东西不太对。管理他的吟游诗人真的是骑士职责范围内的工作吗？在我看来，吟游诗人应该做他份内的事，根本不需要骑士命令他这么做。毕竟，用诗歌记载骑士的探险事迹，这是吟游诗人的职责。为什么骑士还需要提醒吟游诗人去做他份内的事情呢？

此外，因为骑士需要知道吟游诗人，所以就必须把吟游诗人注入到BarveKnight类中。这不仅使BraveKnight的代码复杂化了，而且还让我疑惑是否还需要一个不需要吟游诗人的骑士呢？如果Minstrel为null会发生什么呢？我是否应该引入一个空值校验逻辑来覆盖该场景？

简单的BraveKnight类开始变得复杂，如果你还需要应对没有吟游诗人时的场景，那代码会变得更复杂。但利用AOP，你可以声明吟游诗人必须歌颂骑士的探险事迹，而骑士本身并不用直接访问Minstrel的方法。

要将Minstrel抽象为一个切面，你所需要做的事情就是在一个Spring配置文件中声明它。程序清单1.11是更新后的knights.xml文件，Minstrel被声明为一个切面。

程序清单**1.11**　将**Minstrel**声明为一个切面

```
<?xml version="1.0" encoding="UTF-8"?>>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
    http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="knight" class="com.springinaction.knights.BraveKnight">
    <constructor-arg ref="quest" />
  </bean>

  <bean id="quest" class="com.springinaction.knights.SlayDragonQuest">
    <constructor-arg value="#{T(System).out}" />
  </bean>
  <bean id="minstrel" class="com.springinaction.knights.Minstrel">    声明 Minstrel bean
    <constructor-arg value="#{T(System).out}" />
  </bean>

  <aop:config>
    <aop:aspect ref="minstrel">
      <aop:pointcut id="embark"
          expression="execution(* *.embarkOnQuest(..))"/>    定义切点

      <aop:before pointcut-ref="embark"    声明前置通知
          method="singBeforeQuest"/>

      <aop:after pointcut-ref="embark"    声明后置通知
          method="singAfterQuest"/>
    </aop:aspect>
  </aop:config>

</beans>
```

这里使用了Spring的aop配置命名空间把Minstrel bean声明为一个切面。首先，需要把Minstrel声明为一个bean，然后在<aop:aspect>元素中引用该bean。为了进一步定义切面，声明（使用<aop:before>）在embarkOnQuest()方法执行前调用Minstrel的singBeforeQuest()方法。这种方式被称为前置通知（before advice）。同时声明（使用<aop:after>）在embarkOnQuest()方法执行后调用singAfter Quest()方法。这种方式被称为后置通知（after advice）。

在这两种方式中，pointcut-ref属性都引用了名字为embank的切入点。该切入点是在前边的<pointcut>元素中定义的，并配置expression属性来选择所应用的通知。表达式的语法采用的是AspectJ的切点表达式语言。

现在，你无需担心不了解AspectJ或编写AspectJ切点表达式的细节，我们稍后会在第4章详细地探讨Spring AOP的内容。现在你已经知

道，Spring在骑士执行探险任务前后会调用`Minstrel`的`singBeforeQuest()`和`singAfterQuest()`方法，这就足够了。

这就是我们需要做的所有的事情！通过少量的XML配置，就可以把`Minstrel`声明为一个Spring切面。如果你现在还没有完全理解，不必担心，在第4章你会看到更多的Spring AOP示例，那将会帮助你彻底弄清楚。现在我们可以从这个示例中获得两个重要的观点。

首先，`Minstrel`仍然是一个POJO，没有任何代码表明它要被作为一个切面使用。当我们按照上面那样进行配置后，在Spring的上下文中，`Minstrel`实际上已经变成一个切面了。

其次，也是最重要的，`Minstrel`可以被应用到`BraveKnight`中，而`BraveKnight`不需要显式地调用它。实际上，`BraveKnight`完全不知道`Minstrel`的存在。

必须还要指出的是，尽管我们使用Spring魔法把`Minstrel`转变为一个切面，但首先要把它声明为一个Spring `bean`。能够为其他Spring bean做到的事情都可以同样应用到Spring切面中，例如为它们注入依赖。

应用切面来歌颂骑士可能只是有点好玩而已，但是Spring AOP可以做很多有实际意义的事情。在后续的各章中，你还会了解基于Spring AOP实现声明式事务和安全（第9章和第14章）。

但现在，让我们再看看 Spring简化Java开发的其他方式。

1.1.4 使用模板消除样板式代码

你是否写过这样的代码，当编写的时候总会感觉以前曾经这么写过？我的朋友，这不是似曾相识。这是样板式的代码（boilerplate code）。通常为了实现通用的和简单的任务，你不得不一遍遍地重复编写这样的代码。

遗憾的是，它们中的很多是因为使用Java API而导致的样板式代码。样板式代码的一个常见范例是使用JDBC访问数据库查询数据。举个例子，如果你曾经用过JDBC，那么你或许会写出类似下面的代码。

程序清单**1.12**　许多**Java API**，例如**JDBC**，会涉及编写大量的样板式代码

```
public Employee getEmployeeById(long id) {
  Connection conn = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;
  try {
    conn = dataSource.getConnection();
    stmt = conn.prepareStatement(
          "select id, firstname, lastname, salary from " +
          "employee where id=?");          <-- 查找员工
     stmt.setLong(1, id);
    rs = stmt.executeQuery();
    Employee employee = null;
    if (rs.next()) {
      employee = new Employee();          <-- 根据数据创建对象
      employee.setId(rs.getLong("id"));
      employee.setFirstName(rs.getString("firstname"));
      employee.setLastName(rs.getString("lastname"));
      employee.setSalary(rs.getBigDecimal("salary"));
    }
    return employee;
  } catch (SQLException e) {          <-- 这里应该做什么?
  } finally {
      if(rs != null) {          <-- 清理
        try {
          rs.close();
        } catch(SQLException e) {}
      }
      if(stmt != null) {
        try {
        stmt.close();
        } catch(SQLException e) {}
      }

      if(conn != null) {
        try {
          conn.close();
        } catch(SQLException e) {}
      }
  }
  return null;
}
```

正如你所看到的，这段JDBC代码查询数据库获得员工姓名和薪水。我打赌你很难把上面的代码逐行看完，这是因为少量查询员工的代码淹没在一堆JDBC的样板式代码中。首先你需要创建一个数据库连接，然后再创建一个语句对象，最后你才能进行查询。为了平息JDBC可能会出现的怒火，你必须捕捉`SQLException`，这是一个检查型异常，即使它抛出后你也做不了太多事情。

最后，毕竟该说的也说了，该做的也做了，你不得不清理战场，关闭数据库连接、语句和结果集。同样为了平息JDBC可能会出现的怒火，你依然要捕捉`SQLException`。

程序清单1.12中的代码和你实现其他JDBC操作时所写的代码几乎是

相同的。只有少量的代码与查询员工逻辑有关系，其他的代码都是JDBC的样板代码。

JDBC不是产生样板式代码的唯一场景。在许多编程场景中往往都会导致类似的样板式代码，JMS、JNDI和使用REST服务通常也涉及大量的重复代码。

Spring旨在通过模板封装来消除样板式代码。Spring的JdbcTemplate使得执行数据库操作时，避免传统的JDBC样板代码成为了可能。

举个例子，使用Spring的`JdbcTemplate`（利用了 Java 5特性的`JdbcTemplate`实现）重写的`getEmployeeById()`方法仅仅关注于获取员工数据的核心逻辑，而不需要迎合JDBC API的需求。程序清单1.13展示了修订后的`getEmployeeById()`方法。

程序清单**1.13**　模板能够让你的代码关注于自身的职责

```
public Employee getEmployeeById(long id) {
  return jdbcTemplate.queryForObject(
          "select id, firstname, lastname, salary " +     <—SQL 查询
          "from employee where id=?",
          new RowMapper<Employee>() {
            public Employee mapRow(ResultSet rs,
                    int rowNum) throws SQLException {     <—将结果匹配为对象
              Employee employee = new Employee();
              employee.setId(rs.getLong("id"));
              employee.setFirstName(rs.getString("firstname"));
              employee.setLastName(rs.getString("lastname"));
              employee.setSalary(rs.getBigDecimal("salary"));
              return employee;

            }
          },
          id);    <— 指定查询参数
}
```

正如你所看到的，新版本的`getEmployeeById()`简单多了，而且仅仅关注于从数据库中查询员工。模板的`queryForObject()`方法需要一个SQL查询语句，一个`RowMapper`对象（把数据映射为一个域对象），零个或多个查询参数。`GetEmp loyeeById()`方法再也看不到以前的JDBC样板式代码了，它们全部被封装到了模板中。

我已经向你展示了Spring通过面向POJO编程、DI、切面和模板技术来简化Java开发中的复杂性。在这个过程中，我展示了在基于XML的配置文件中如何配置bean和切面，但这些文件是如何加载的呢？它们被加载到哪里去了？让我们再了解下Spring容器，这是应用中的所有bean所驻留的地方。

1.2　容纳你的Bean

在基于Spring的应用中，你的应用对象生存于Spring容器（container）中。如图1.4所示，Spring容器负责创建对象，装配它们，配置它们并管理它们的整个生命周期，从生存到死亡（在这里，可能就是`new`到`finalize()`）。

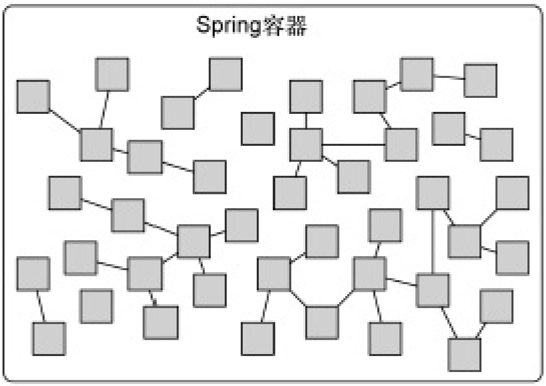

图1.4　在Spring应用中，对象由Spring容器创建和装配，并存在容器之中

在下一章，你将了解如何配置Spring，从而让它知道该创建、配置和组装哪些对象。但首先，最重要的是了解容纳对象的容器。理解容器将有助于理解对象是如何被管理的。

容器是Spring框架的核心。Spring容器使用DI管理构成应用的组件，它会创建相互协作的组件之间的关联。毫无疑问，这些对象更简单干净，更易于理解，更易于重用并且更易于进行单元测试。

Spring容器并不是只有一个。Spring自带了多个容器实现，可以归为两种不同的类型。bean工厂（由`org.springframework. beans.factory.eanFactory`接口定义）是最简单的容器，提供基本的DI支持。应用上下文（由`org.springframework.context.ApplicationContext`接口定义）基于BeanFactory构建，并提供应用框架级别的服务，例如从属性文件解析文本信息以及发布应用事件给感兴趣的事件监听者。

虽然我们可以在bean工厂和应用上下文之间任选一种，但bean工厂对大多数应用来说往往太低级了，因此，应用上下文要比bean工厂更受欢迎。我们会把精力集中在应用上下文的使用上，不再浪费时间讨论bean工厂。

1.2.1 使用应用上下文

Spring自带了多种类型的应用上下文。下面罗列的几个是你最有可能遇到的。

- `AnnotationConfigApplicationContext`：从一个或多个基于Java的配置类中加载Spring应用上下文。
- `AnnotationConfigWebApplicationContext`：从一个或多个基于Java的配置类中加载Spring Web应用上下文。
- `ClassPathXmlApplicationContext`：从类路径下的一个或多个XML配置文件中加载上下文定义，把应用上下文的定义文件作为类资源。
- `FileSystemXmlapplicationcontext`：从文件系统下的一个或多个XML配置文件中加载上下文定义。
- `XmlWebApplicationContext`：从Web应用下的一个或多个XML配置文件中加载上下文定义。

当在第8章讨论基于Web的Spring应用时，我们会对`AnnotationConfigWeb-ApplicationContext`和`XmlWebApplicationContext`进行更详细的讨论。现在我们先简单地使用`FileSystemXmlApplicationContext`从文件系统中加载应用上下文或者使用`ClassPathXmlApplicationContext`从类路径中加载应用上下文。

无论是从文件系统中装载应用上下文还是从类路径下装载应用上下文，将bean加载到bean工厂的过程都是相似的。例如，如下代码展示了如何加载一个`FileSystemXmlApplicationContext`：

```
ApplicationContext context = new
        FileSystemXmlApplicationContext("c:/knight.xml");
```

类似地，你可以使用`ClassPathXmlApplicationContext`从应用的类路径下加载应用上下文：

```
ApplicationContext context = new
        ClassPathXmlApplicationContext("knight.xml");
```

使用`FileSystemXmlApplicationContext`和使用`ClassPathXmlApp-licationContext`的区别在于：`FileSystemXmlApplicationContext`在指定的文件系统路径下查找knight.xml文件；而`ClassPathXmlApplicationContext`是在所有的类路径（包含JAR文件）下查找 knight.xml文件。

如果你想从Java配置中加载应用上下文，那么可以使用AnnotationConfig-ApplicationContext：

```
ApplicationContext context = new AnnotationConfigApplicationContext(
    com.springinaction.knights.config.KnightConfig.class);
```

在这里没有指定加载Spring应用上下文所需的XML文件，AnnotationConfig-ApplicationContext通过一个配置类加载bean。

应用上下文准备就绪之后，我们就可以调用上下文的getBean()方法从Spring容器中获取bean。

现在你应该基本了解了如何创建Spring容器，让我们对容器中bean的生命周期做更进一步的探究。

1.2.2　bean的生命周期

在传统的Java应用中，bean的生命周期很简单。使用Java关键字new进行bean实例化，然后该bean就可以使用了。一旦该bean不再被使用，则由Java自动进行垃圾回收。

相比之下，Spring容器中的bean的生命周期就显得相对复杂多了。正确理解Spring bean的生命周期非常重要，因为你或许要利用Spring提供的扩展点来自定义bean的创建过程。图1.5展示了bean装载到Spring应用上下文中的一个典型的生命周期过程。

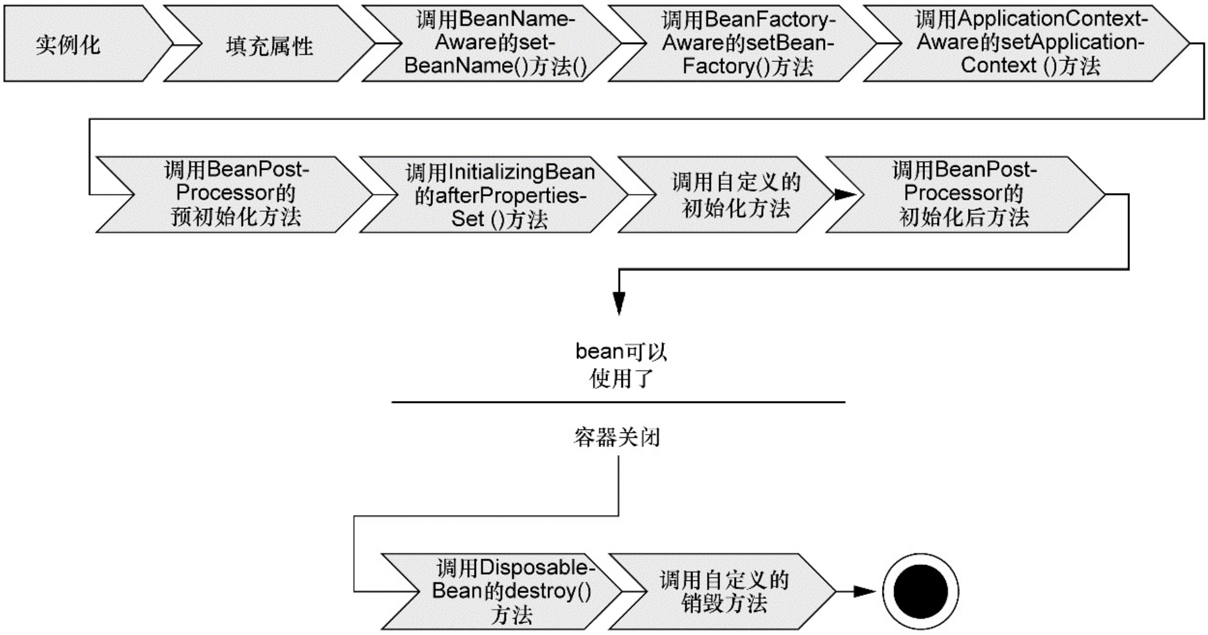

图1.5　bean在Spring容器中从创建到销毁经历了
若干阶段，每一阶段都可以针对Spring如何管理bean进行个性化定制

正如你所见，在bean准备就绪之前，bean工厂执行了若干启动步骤。我们对图1.5进行详细描述：

1．Spring对bean进行实例化；

2．Spring将值和bean的引用注入到bean对应的属性中；

3．如果bean实现了`BeanNameAware`接口，Spring将bean的ID传递给`setBean-Name()`方法；

4．如果bean实现了`BeanFactoryAware`接口，Spring将调用`setBeanFactory()`方法，将BeanFactory容器实例传入；

5．如果bean实现了`ApplicationContextAware`接口，Spring将调用`setApplicationContext()`方法，将bean所在的应用上下文的引用传入进来；

6．如果bean实现了`BeanPostProcessor`接口，Spring将调用它们的`post-ProcessBeforeInitialization()`方法；

7．如果bean实现了`InitializingBean`接口，Spring将调用它们的`after-PropertiesSet()`方法。类似地，如果bean使用`init-method`声明了初始化方法，该方法也会被调用；

8．如果bean实现了`BeanPostProcessor`接口，Spring将调用它们的`post-ProcessAfterInitialization()`方法；

9．此时，bean已经准备就绪，可以被应用程序使用了，它们将一直驻留在应用上下文中，直到该应用上下文被销毁；

10．如果bean实现了`DisposableBean`接口，Spring将调用它的`destroy()`接口方法。同样，如果bean使用`destroy-method`声明了销毁方法，该方法也会被调用。

现在你已经了解了如何创建和加载一个Spring容器。但是一个空的容器并没有太大的价值，在你把东西放进去之前，它里面什么都没有。为了从Spring的DI中受益，我们必须将应用对象装配进Spring容器中。我们将在第2章对bean装配进行更详细的探讨。

我们现在首先浏览一下Spring的体系结构，了解一下Spring框架的基本组成部分和最新版本的Spring所发布的新特性。

1.3 俯瞰Spring风景线

正如你所看到的，Spring框架关注于通过DI、AOP和消除样板式代码来简化企业级Java开发。即使这是Spring所能做的全部事情，那Spring也值得一用。但是，Spring实际上的功能超乎你的想象。

在Spring框架的范畴内，你会发现Spring简化Java开发的多种方式。但在Spring框架之外还存在一个构建在核心框架之上的庞大生态圈，它将Spring扩展到不同的领域，例如Web服务、REST、移动开发以及NoSQL。

首先让我们拆开Spring框架的核心来看看它究竟为我们带来了什么，然后我们再浏览下Spring Portfolio中的其他成员。

1.3.1 Spring模块

当我们下载Spring发布版本并查看其lib目录时，会发现里面有多个JAR文件。在Spring 4.0中，Spring框架的发布版本包括了20个不同的模块，每个模块会有3个JAR文件（二进制类库、源码的JAR文件以及JavaDoc的JAR文件）。完整的库JAR文件如图1.6所示。

图1.6　Spring框架由20个不同的模块组成

这些模块依据其所属的功能可以划分为6类不同的功能，如图1.7所示。

总体而言，这些模块为开发企业级应用提供了所需的一切。但是你也不必将应用建立在整个Spring框架之上，你可以自由地选择适合自身应用需求的Spring模块；当Spring不能满足需求时，完全可以考虑其他选择。事实上，Spring甚至提供了与其他第三方框架和类库的集成点，这样你就不需要自己编写这样的代码了。

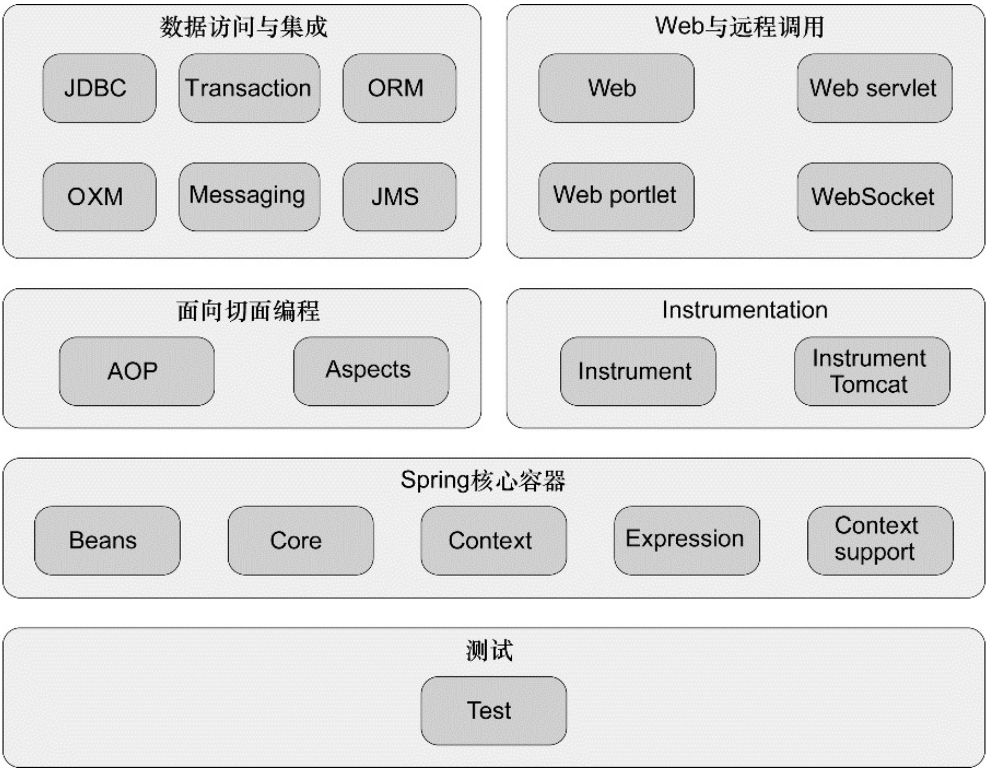

图1.7　Spring框架由6个定义良好的模块分类组成

让我们逐一浏览Spring的模块，看看它们是如何构建起Spring整体蓝图的。

Spring核心容器

容器是Spring框架最核心的部分，它管理着Spring应用中bean的创建、配置和管理。在该模块中，包括了Spring bean工厂，它为Spring提供了DI的功能。基于bean工厂，我们还会发现有多种Spring应用上下文的实现，每一种都提供了配置Spring的不同方式。

除了bean工厂和应用上下文，该模块也提供了许多企业服务，例如E-mail、JNDI访问、EJB集成和调度。

所有的Spring模块都构建于核心容器之上。当你配置应用时，其实你隐式地使用了这些类。贯穿本书，我们都会涉及到核心模块，在第2章中我们将会深入探讨Spring的DI。

Spring的AOP模块

在AOP模块中，Spring对面向切面编程提供了丰富的支持。这个模块是Spring应用系统中开发切面的基础。与DI一样，AOP可以帮助应用

对象解耦。借助于AOP，可以将遍布系统的关注点（例如事务和安全）从它们所应用的对象中解耦出来。

我们将在第4章深入探讨Spring对AOP支持。

数据访问与集成

使用JDBC编写代码通常会导致大量的样板式代码，例如获得数据库连接、创建语句、处理结果集到最后关闭数据库连接。Spring的JDBC和DAO（Data Access Object）模块抽象了这些样板式代码，使我们的数据库代码变得简单明了，还可以避免因为关闭数据库资源失败而引发的问题。该模块在多种数据库服务的错误信息之上构建了一个语义丰富的异常层，以后我们再也不需要解释那些隐晦专有的SQL错误信息了！

对于那些更喜欢ORM（Object-Relational Mapping）工具而不愿意直接使用JDBC的开发者，Spring提供了ORM模块。Spring的ORM模块建立在对DAO的支持之上，并为多个ORM框架提供了一种构建DAO的简便方式。Spring没有尝试去创建自己的ORM解决方案，而是对许多流行的ORM框架进行了集成，包括Hibernate、Java Persisternce API、Java Data Object和iBATIS SQL Maps。Spring的事务管理支持所有的ORM框架以及JDBC。

在第10章讨论Spring数据访问时，你会看到Spring基于模板的JDBC抽象层能够极大地简化JDBC代码。

本模块同样包含了在JMS（Java Message Service）之上构建的Spring抽象层，它会使用消息以异步的方式与其他应用集成。从Spring 3.0开始，本模块还包含对象到XML映射的特性，它最初是Spring Web Service项目的一部分。

除此之外，本模块会使用Spring AOP模块为Spring应用中的对象提供事务管理服务。

Web与远程调用

MVC（Model-View-Controller）模式是一种普遍被接受的构建Web应用的方法，它可以帮助用户将界面逻辑与应用逻辑分离。Java从来不缺少MVC框架，Apache的Struts、JSF、WebWork和Tapestry都是可选的最流行的MVC框架。

虽然Spring能够与多种流行的MVC框架进行集成，但它的Web和远程调用模块自带了一个强大的MVC框架，有助于在Web层提升应用的松耦合水平。在第5章到第7章中，我们将会学习Spring的MVC框架。

除了面向用户的Web应用，该模块还提供了多种构建与其他应用交互的远程调用方案。Spring远程调用功能集成了RMI（Remote Method Invocation）、Hessian、Burlap、JAX-WS，同时Spring还自带了一个远程调用框架：HTTP invoker。Spring还提供了暴露和使用REST API的良好支持。

我们将会在第15章讨论Spring的远程调用功能。在第16章学习如何创建和使用REST API。

Instrumentation

Spring的Instrumentation模块提供了为JVM添加代理（agent）的功能。具体来讲，它为Tomcat提供了一个织入代理，能够为Tomcat传递类文件，就像这些文件是被类加载器加载的一样。

如果这听起来有点难以理解，不必对此过于担心。这个模块所提供的Instrumentation使用场景非常有限，在本书中，我们不会介绍该模块。

测试

鉴于开发者自测的重要性，Spring提供了测试模块以致力于Spring应用的测试。

通过该模块，你会发现Spring为使用JNDI、Servlet和Portlet编写单元测试提供了一系列的mock对象实现。对于集成测试，该模块为加载Spring应用上下文中的bean集合以及与Spring上下文中的bean进行交互提供了支持。

在本书中，有很多的样例都是测试驱动的，将会使用到Spring所提供的测试功能。

1.3.2 Spring Portfolio

当谈论Spring时，其实它远远超出我们的想象。事实上，Spring远不是Spring框架所下载的那些。如果仅仅停留在核心的Spring框架层面，我们将错过Spring Portfolio所提供的巨额财富。整个Spring Portfolio包括多个构建于核心Spring框架之上的框架和类库。概括地

讲，整个Spring Portfolio几乎为每一个领域的Java开发都提供了Spring编程模型。

或许需要几卷书才能覆盖Spring Portfolio所提供的所有内容，这也远远超出了本书的范围。不过，我们会介绍Spring Portfolio中的一些项目，同样，我们将体验一下核心框架之外的另一番风景。

Spring Web Flow

Spring Web Flow建立于Spring MVC框架之上，它为基于流程的会话式Web应用（可以想一下购物车或者向导功能）提供了支持。我们将在第8章讨论更多关于Spring Web Flow的内容，你还可以访问Spring Web Flow的主页（http://projects.spring.io/spring-webflow/）。

Spring Web Service

虽然核心的Spring框架提供了将Spring bean以声明的方式发布为Web Service的功能，但是这些服务是基于一个具有争议性的架构（拙劣的契约后置模型）之上而构建的。这些服务的契约由bean的接口来决定。 Spring Web Service提供了契约优先的Web Service模型，服务的实现都是为了满足服务的契约而编写的。

本书不会再探讨Spring Web Service，但是你可以浏览站点http://docs.spring.io/spring- ws/site/来了解更多关于Spring Web Service的信息。

Spring Security

安全对于许多应用都是一个非常关键的切面。利用Spring AOP，Spring Security为Spring应用提供了声明式的安全机制。你将会在第9章看到如何为应用的Web层添加Spring Security功能。同时，我们还会在第14章重新回到Spring Security的话题，学习如何保护方法调用。你可以在主页http://projects.spring.io/spring-security/上获得关于Spring Security的更多信息。

Spring Integration

许多企业级应用都需要与其他应用进行交互。Spring Integration提供了多种通用应用集成模式的Spring声明式风格实现。

我们不会在本书覆盖Spring Integration的内容，但是如果你想了解更多

关于Spring Integration的信息，我推荐Mark Fisher、Jonas Partner、Marius Bogoevici和Iwein Fuld编写的《Spring Integration in Action》（Manning，2012，www.manning.com/fisher/）；或者你还可以访问Spring Integration的主页http://projects.spring.io/spring-integration/。

Spring Batch

当我们需要对数据进行大量操作时，没有任何技术可以比批处理更胜任这种场景。如果需要开发一个批处理应用，你可以通过Spring Batch，使用Spring强大的面向POJO的编程模型。

Spring Batch超出了本书的范畴，但是你可以阅读Arnaud Cogoluegnes、Thierry Templier、Gary Gregory和Olivier Bazoud编写的《Spring Batch in Action》（Manning，2012，www.manning.com/templier/），或者访问Spring Batch的主页http://projects.spring.io/ spring-batch/。

Spring Data

Spring Data使得在Spring中使用任何数据库都变得非常容易。尽管关系型数据库统治企业级应用多年，但是现代化的应用正在认识到并不是所有的数据都适合放在一张表中的行和列中。一种新的数据库种类，通常被称之为*NoSQL*数据库[2]，提供了使用数据的新方法，这些方法会比传统的关系型数据库更为合适。

不管你使用文档数据库，如MongoDB，图数据库，如Neo4j，还是传统的关系型数据库，Spring Data都为持久化提供了一种简单的编程模型。这包括为多种数据库类型提供了一种自动化的Repository机制，它负责为你创建Repository的实现。

我们将会在第11章看到如何使用Spring Data简化Java Persistence API（JPA）开发，然后在第12章，将相关的讨论拓展至几种NoSQL数据库。

Spring Social

社交网络是互联网领域中新兴的一种潮流，越来越多的应用正在融入社交网络网站，例如Facebook或者Twitter。如果对此感兴趣，你可以了解一下Spring Social，这是Spring的一个社交网络扩展模块。

不过，Spring Social并不仅仅是tweet和好友。尽管名字是这样，但

Spring Social更多的是关注连接（connect），而不是社交（social）。它能够帮助你通过REST API连接Spring应用，其中有些Spring应用可能原本并没有任何社交方面的功能目标。

限于篇幅，我们在本书中不会涉及Spring Social。但是，如果你对Spring如何帮助你连接Facebook或Twitter感兴趣的话，可以查看网址https://spring.io/guides/gs/accessing- facebook/和https://spring.io/guides/gs/accessing-twitter/中的入门指南。

Spring Mobile

移动应用是另一个引人瞩目的软件开发领域。智能手机和平板设备已成为许多用户首选的客户端。Spring Mobile是Spring MVC新的扩展模块，用于支持移动Web应用开发。

Spring for Android

与Spring Mobile相关的是Spring Android项目。这个新项目，旨在通过Spring框架为开发基于Android设备的本地应用提供某些简单的支持。最初，这个项目提供了Spring `RestTemplate`的一个可以用于Android应用之中的版本。它还能与Spring Social协作，使得原生应用可以通过REST API进行社交网络的连接。

本书中，我不会讨论Spring for Android，不过你可以通过http://projects.spring.io /spring-android/了解更多内容。

Spring Boot

Spring极大地简化了众多的编程任务，减少甚至消除了很多样板式代码，如果没有Spring的话，在日常工作中你不得不编写这样的样板代码。Spring Boot是一个崭新的令人兴奋的项目，它以Spring的视角，致力于简化Spring本身。

Spring Boot大量依赖于自动配置技术，它能够消除大部分（在很多场景中，甚至是全部）Spring配置。它还提供了多个Starter项目，不管你使用Maven还是Gradle，这都能减少Spring工程构建文件的大小。

在本书即将结束的第21章，我们将会学习Spring Boot。

1.4　Spring的新功能

当本书的第3版交付印刷的时候，当时Spring的最新版本是3.0.5。那大约是在3年前，从那时到现在发生了很多的变化。Spring框架经历了3个重要的发布版本——3.1、3.2以及现在的4.0——每个版本都带来了新的特性和增强，以简化应用程序的研发。Spring Portfolio中的一些成员项目也经历了重要的变更。

本书也进行了更新，试图涵盖这些发布版本中众多最令人兴奋和有用的特性。但现在，我们先简要地了解一下Spring带来了哪些新功能。

1.4.1 Spring 3.1新特性

Spring 3.1带来了多项有用的新特性和增强，其中有很多都是关于如何简化和改善配置的。除此之外，Spring 3.1还提供了声明式缓存的支持以及众多针对Spring MVC的功能增强。下面的列表展现了Spring 3.1重要的功能升级：

- 为了解决各种环境下（如开发、测试和生产）选择不同配置的问题，Spring 3.1引入了环境profile功能。借助于profile，就能根据应用部署在什么环境之中选择不同的数据源bean；
- 在Spring 3.0基于Java的配置之上，Spring 3.1添加了多个enable注解，这样就能使用这个注解启用Spring的特定功能；
- 添加了Spring对声明式缓存的支持，能够使用简单的注解声明缓存边界和规则，这与你以前声明事务边界很类似；
- 新添加的用于构造器注入的c命名空间，它类似于Spring 2.0所提供的面向属性的p命名空间，p命名空间用于属性注入，它们都是非常简洁易用的；
- Spring开始支持Servlet 3.0，包括在基于Java的配置中声明Servlet和Filter，而不再借助于web.xml；
- 改善Spring对JPA的支持，使得它能够在Spring中完整地配置JPA，不必再使用persistence.xml文件。

Spring 3.1还包含了多项针对Spring MVC的功能增强：

- 自动绑定路径变量到模型属性中；
- 提供了`@RequestMappingproduces`和`consumes`属性，用于匹配请求中的`Accept`和`Content-Type`头部信息；
- 提供了`@RequestPart`注解，用于将multipart请求中的某些部分绑定到处理器的方法参数中；
- 支持Flash属性（在redirect请求之后依然能够存活的属性）以及用于在请求间存放flash属性的`RedirectAttributes`类型。

除了Spring 3.1所提供的新功能以外，同等重要的是要注意Spring 3.1不再支持的功能。具体来讲，为了支持原生的EntityManager，Spring的`JpaTemplate`和`JpaDaoSupport`类被废弃掉了。尽管它们已经被废弃了，但直到Spring 3.2版本，它依然是可以使用的。但最好不要再使用它们了，因为它们不会进行更新以支持JPA 2.0，并且已经在Spring 4中移除掉了。

现在，让我们看一下Spring 3.2提供了什么新功能。

1.4.2 Spring 3.2新特性

Spring 3.1在很大程度上聚焦于配置改善以及其他的一些增强，包括Spring MVC的增强，而Spring 3.2是主要关注Spring MVC的一个发布版本。Spring MVC 3.2带来了如下的功能提升：

- Spring 3.2的控制器（Controller）可以使用Servlet 3.0的异步请求，允许在一个独立的线程中处理请求，从而将Servlet线程解放出来处理更多的请求；
- 尽管从Spring 2.5开始，Spring MVC控制器就能以POJO的形式进行很便利地测试，但是Spring 3.2引入了Spring MVC测试框架，用于为控制器编写更为丰富的测试，断言它们作为控制器的行为行为是否正确，而且在使用的过程中并不需要Servlet容器；
- 除了提升控制器的测试功能，Spring 3.2还包含了基于`RestTemplate`的客户端的测试支持，在测试的过程中，不需要往真正的REST端点上发送请求；
- `@ControllerAdvice`注解能够将通用的`@ExceptionHandler`、`@ InitBinder`和`@ModelAttributes`方法收集到一个类中，并应用到所有控制器上；
- 在Spring 3.2之前，只能通过`ContentNegotiatingViewResolver`使用完整的内容协商（full content negotiation）功能。但是在Spring 3.2中，完整的内容协商功能可以在整个Spring MVC中使用，即便是依赖于消息转换器（message converter）使用和产生内容的控制器方法也能使用该功能；
- Spring MVC 3.2包含了一个新的`@MatrixVariable`注解，这个注解能够将请求中的矩阵变量（matrix variable）绑定到处理器的方法参数中；
- 基础的抽象类`AbstractDispatcherServletInitializer`能够非常便利地配置`DispatcherServlet`，而不必再使用

web.xml。与之类似，当你希望通过基于Java的方式来配置Spring的时候，可以使用Abstract- AnnotationConfigDispatcherServletInitializer的子类；
- 新增了ResponseEntityExceptionHandler，可以用来替代Default- HandlerExceptionResolver。ResponseEntityExceptionHandler方法会返回ResponseEntity<Object>，而不是ModelAndView；
- RestTemplate和@RequestBody的参数可以支持范型；
- RestTemplate和@RequestMapping可以支持HTTP PATCH方法；
- 在拦截器匹配时，支持使用URL模式将其排除在拦截器的处理功能之外。

虽然Spring MVC是Spring 3.2改善的核心内容，但是它依然还增加了多项非MVC的功能改善。下面列出了Spring 3.2中几项最为有意思的新特性：

- @Autowired、@Value和@Bean注解能够作为元注解，用于创建自定义的注入和bean声明注解；
- @DateTimeFormat注解不再强依赖JodaTime。如果提供了JodaTime，就会使用它，否则的话，会使用SimpleDateFormat；
- Spring的声明式缓存提供了对JCache 0.5的支持；
- 支持定义全局的格式来解析和渲染日期与时间；
- 在集成测试中，能够配置和加载WebApplicationContext；
- 在集成测试中，能够针对request和session作用域的bean进行测试。

在本书的多个章节中，都能看到Spring 3.2的特性，尤其是在Web和REST相关的章节中。

1.4.3 Spring 4.0新特性

当编写本书时，Spring 4.0是最新的发布版本。在Spring 4.0中包含了很多令人兴奋的新特性，包括：

- Spring提供了对WebSocket编程的支持，包括支持JSR-356——Java API for WebSocket；
- 鉴于WebSocket仅仅提供了一种低层次的API，急需高层次的抽象，因此Spring 4.0在WebSocket之上提供了一个高层次的面向消

息的编程模型，该模型基于SockJS，并且包含了对STOMP协议的支持；

- 新的消息（messaging）模块，很多的类型来源于Spring Integration项目。这个消息模块支持Spring的SockJS/STOMP功能，同时提供了基于模板的方式发布消息；
- Spring是第一批（如果不说是第一个的话）支持Java 8特性的Java框架，比如它所支持的lambda表达式。别的暂且不说，这首先能够让使用特定的回调接口（如`RowMapper`和`JdbcTemplate`）更加简洁，代码更加易读；
- 与Java 8同时得到支持的是JSR-310——Date与Time API，在处理日期和时间时，它为开发者提供了比`java.util.Date`或`java.util.Calendar`更丰富的API；
- 为Groovy开发的应用程序提供了更加顺畅的编程体验，尤其是支持非常便利地完全采用Groovy开发Spring应用程序。随这些一起提供的是来自于Grails的BeanBuilder，借助它能够通过Groovy配置Spring应用；
- 添加了条件化创建bean的功能，在这里只有开发人员定义的条件满足时，才会创建所声明的bean；
- Spring 4.0包含了Spring `RestTemplate`的一个新的异步实现，它会立即返回并且允许在操作完成后执行回调；
- 添加了对多项JEE规范的支持，包括JMS 2.0、JTA 1.2、JPA 2.1和Bean Validation 1.1。

可以看到，在Spring框架的最新发布版本中，包含了很多令人兴奋的新特性。在本书中，我们将会看到很多这样的新特性，同时也会学习Spring中长期以来一直存在的特性。

1.5 小结

现在，你应该对Spring的功能特性有了一个清晰的认识。Spring致力于简化企业级Java开发，促进代码的松散耦合。成功的关键在于依赖注入和AOP。

在本章，我们先体验了Spring的DI。DI是组装应用对象的一种方式，借助这种方式对象无需知道依赖来自何处或者依赖的实现方式。不同于自己获取依赖对象，对象会在运行期赋予它们所依赖的对象。依赖对象通常会通过接口了解所注入的对象，这样的话就能确保低耦合。

除了DI，我们还简单介绍了Spring对AOP的支持。AOP可以帮助应用将散落在各处的逻辑汇集于一处——切面。当Spring装配bean的时

候，这些切面能够在运行期编织起来，这样就能非常有效地赋予bean新的行为。

依赖注入和AOP是Spring框架最核心的部分，因此只有理解了如何应用Spring最关键的功能，你才有能力使用Spring框架的其他功能。在本章，我们只是触及了Spring DI和AOP特性的皮毛。在以后的几章，我们将深入探讨DI和AOP。

闲言少叙，我们立即转到第2章学习如何在Spring中使用DI装配对象。

---

[1]对于基于Java的配置，Spring提供了AnnotationConfigApplicationContext。

[2]相对于NoSQL，我更喜欢非关系型（non-relational）或无模式（schema-less）这样的术语。将这些数据库称之为NoSQL，实际上将问题归因于查询语言，而不是数据模型。

第2章　装配Bean

本章内容：

- 声明bean
- 构造器注入和Setter方法注入
- 装配bean
- 控制bean的创建和销毁

在看电影的时候，你曾经在电影结束后留在位置上继续观看片尾字幕吗？一部电影需要由这么多人齐心协力才能制作出来，这真是有点令人难以置信！除了主要的参与人员——演员、编剧、导演和制片人，还有那些幕后人员——音乐师、特效制作人员和艺术指导，更不用说道具师、录音师、服装师、化妆师、特技演员、广告师、第一助理摄影师、第二助理摄影师、布景师、灯光师和伙食管理员（或许是最重要的人员）了。

现在想象一下，如果这些人彼此之间没有任何交流，你最喜爱的电影会变成什么样子？让我这么说吧，他们都出现在摄影棚中，开始各做各的事情，彼此之间互不合作。如果导演保持沉默不喊“开机”，摄影师就不会开始拍摄。或许这并没什么大不了的，因为女主角还呆在她的保姆车里，而且因为没有雇佣灯光师，一切处于黑暗之中。或许你曾经看过类似这样的电影。但是大多数电影（总之，都还是很优秀的）都是由成千上万的人一起协作来完成的，他们有着共同的目标：制作一部广受欢迎的佳作。

在这方面，一个优秀的软件与之相比并没有太大区别。任何一个成功的应用都是由多个为了实现某一个业务目标而相互协作的组件构成的。这些组件必须彼此了解，并且相互协作来完成工作。例如，在一个在线购物系统中，订单管理组件需要和产品管理组件以及信用卡认证组件协作。这些组件或许还需要与数据访问组件协作，从数据库读取数据以及把数据写入数据库。

但是，正如我们在第1章中所看到的，创建应用对象之间关联关系的传统方法（通过构造器或者查找）通常会导致结构复杂的代码，这些代码很难被复用也很难进行单元测试。如果情况不严重的话，这些对象所做的事情只是超出了它应该做的范围；而最坏的情况则是，这些对象彼此之间高度耦合，难以复用和测试。

在Spring中，对象无需自己查找或创建与其所关联的其他对象。相反，容器负责把需要相互协作的对象引用赋予各个对象。例如，一个订单管理组件需要信用卡认证组件，但它不需要自己创建信用卡认证组件。订单管理组件只需要表明自己两手空空，容器就会主动赋予它一个信用卡认证组件。

创建应用对象之间协作关系的行为通常称为装配（wiring），这也是依赖注入（DI）的本质。在本章我们将介绍使用Spring装配 bean的基础知识。因为DI是Spring的最基本要素，所以在开发基于Spring的应用时，你随时都在使用这些技术。

在Spring中装配bean有多种方式。作为本章的开始，我们先花一点时间来介绍一下配置Spring容器最常见的三种方法。

2.1 Spring配置的可选方案

如第1章中所述，Spring容器负责创建应用程序中的bean并通过DI来协调这些对象之间的关系。但是，作为开发人员，你需要告诉Spring要创建哪些bean并且如何将其装配在一起。当描述bean如何进行装配时，Spring具有非常大的灵活性，它提供了三种主要的装配机制：

- 在XML中进行显式配置。
- 在Java中进行显式配置。
- 隐式的bean发现机制和自动装配。

乍看上去，提供三种可选的配置方案会使Spring变得复杂。每种配置技术所提供的功能会有一些重叠，所以在特定的场景中，确定哪种技术最为合适就会变得有些困难。但是，不必紧张——在很多场景下，选择哪种方案很大程度上就是个人喜好的问题，你尽可以选择自己最喜欢的方式。

Spring有多种可选方案来配置bean，这是非常棒的，但有时候你必须要在其中做出选择。

这方面，并没有唯一的正确答案。你所做出的选择必须要适合你和你的项目。而且，谁说我们只能选择其中的一种方案呢？Spring的配置风格是可以互相搭配的，所以你可以选择使用XML装配一些bean，使用Spring基于Java的配置（JavaConfig）来装配另一些bean，而将剩余的bean让Spring去自动发现。

即便如此，我的建议是尽可能地使用自动配置的机制。显式配置越少越好。当你必须要显式配置bean的时候（比如，有些源码不是由你来维护的，而当你需要为这些代码配置bean的时候），我推荐使用类型安全并且比XML更加强大的JavaConfig。最后，只有当你想要使用便利的XML命名空间，并且在JavaConfig中没有同样的实现时，才应该使用XML。

在本章中，我们会详细介绍这三种技术并且在整本书中都会用到它们。现在，我们会尝试一下每种方法，对它们是什么样子的有一个直观的印象。作为Spring配置的开始，我们先看一下Spring的自动化配置。

2.2 自动化装配bean

在本章稍后的内容中，你会看到如何借助Java和XML来进行Spring装配。尽管你会发现这些显式装配技术非常有用，但是在便利性方面，最强大的还是Spring的自动化配置。如果Spring能够进行自动化装配的话，那何苦还要显式地将这些bean装配在一起呢？

Spring从两个角度来实现自动化装配：

- 组件扫描（component scanning）：Spring会自动发现应用上下文中所创建的bean。
- 自动装配（autowiring）：Spring自动满足bean之间的依赖。

组件扫描和自动装配组合在一起就能发挥出强大的威力，它们能够将你的显式配置降低到最少。

为了阐述组件扫描和装配，我们需要创建几个bean，它们代表了一个音响系统中的组件。首先，要创建`CompactDisc`类，Spring会发现它并将其创建为一个bean。然后，会创建一个`CDPlayer`类，让Spring发现它，并将`CompactDisc`bean注入进来。

2.2.1 创建可被发现的bean

在这个MP3和流式媒体音乐的时代，CD（compact disc）显得有点典雅甚至陈旧。它不像卡带机、八轨磁带、塑胶唱片那么普遍，随着以物理载体进行音乐交付的方式越来越少，CD也变得越来越稀少了。

尽管如此，CD为我们阐述DI如何运行提供了一个很好的样例。如果

你不将CD插入（注入）到CD播放器中，那么CD播放器其实是没有太大用处的。所以，可以这样说，CD播放器依赖于CD才能完成它的使命。

为了在Spring中阐述这个例子，让我们首先在Java中建立CD的概念。程序清单2.1展现了CompactDisc，它是定义CD的一个接口：

程序清单**2.1　CompactDisc**接口在**Java**中定义了**CD**的概念

```
package soundsystem;

public interface CompactDisc {
  void play();
}
```

CompactDisc的具体内容并不重要，重要的是你将其定义为一个接口。作为接口，它定义了CD播放器对一盘CD所能进行的操作。它将CD播放器的任意实现与CD本身的耦合降低到了最小的程度。

我们还需要一个CompactDisc的实现，实际上，我们可以有CompactDisc接口的多个实现。在本例中，我们首先会创建其中的一个实现，也就是程序清单2.2所示的SgtPeppers类。

程序清单**2.2**　带有**@Component**注解的**CompactDisc**实现类**SgtPeppers**

```
package soundsystem;
import org.springframework.stereotype.Component;

@Component
public class SgtPeppers implements CompactDisc {

  private String title = "Sgt. Pepper's Lonely Hearts Club Band";
  private String artist = "The Beatles";

  public void play() {
    System.out.println("Playing " + title + " by " + artist);
  }

}
```

和CompactDisc接口一样，SgtPeppers的具体内容并不重要。你需要注意的就是SgtPeppers类上使用了@Component注解。这个简单的注解表明该类会作为组件类，并告知Spring要为这个类创建bean。没有必要显式配置SgtPeppersbean，因为这个类使用了@Component注解，所以Spring会为你把事情处理妥当。

不过，组件扫描默认是不启用的。我们还需要显式配置一下Spring，从而命令它去寻找带有@Component注解的类，并为其创建bean。程序清单2.3的配置类展现了完成这项任务的最简洁配置。

程序清单**2.3　@ComponentScan**注解启用了组件扫描

```
package soundsystem;
import org.springframework.context.annotation.componentScan;
import org.springframework.context.annotation.Con

@Configuration
@ComponentScan
public class CDPlayerConfig {
}
```

类CDPlayerConfig通过Java代码定义了Spring的装配规则。在2.3节中，我们还会更为详细地介绍基于Java的Spring配置。不过，现在我们只需观察一下CDPlayerConfig类并没有显式地声明任何bean，只不过它使用了@ComponentScan注解，这个注解能够在Spring中启用组件扫描。

如果没有其他配置的话，@ComponentScan默认会扫描与配置类相同的包。因为CDPlayerConfig类位于soundsystem包中，因此Spring将会扫描这个包以及这个包下的所有子包，查找带有@Component注解的类。这样的话，就能发现CompactDisc，并且会在Spring中自动为其创建一个bean。

如果你更倾向于使用XML来启用组件扫描的话，那么可以使用Spring context命名空间的<context:component-scan>元素。程序清单2.4展示了启用组件扫描的最简洁XML配置。

程序清单**2.4**　通过**XML**启用组件扫描

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:Context ="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context>
    http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="soundsystem"/>

</beans>
```

尽管我们可以通过XML的方案来启用组件扫描，但是在后面的讨论中，我更多的还是会使用基于Java的配置。如果你更喜欢XML的话，<context:component-scan>元素会有与@ComponentScan注解相对应的属性和子元素。

可能有点让人难以置信，我们只创建了两个类，就能对功能进行一番尝试了。为了测试组件扫描的功能，我们创建一个简单的JUnit测试，它会创建Spring上下文，并判断CompactDisc是不是真的创建出来了。程序清单2.5中的CDPlayerTest就是用来完成这项任务的。

程序清单**2.5**　测试组件扫描能够发现**CompactDisc**

```
package soundsystem;

import static org.junit.Assert.*;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=CDPlayerConfig.class)
public class CDPlayerTest {

  @Autowired
  private CompactDisc cd;

  @Test
  public void cdShouldNotBeNull() {
    assertNotNull(cd);
  }

}
```

CDPlayerTest使用了Spring的SpringJUnit4ClassRunner，以便在测试开始的时候自动创建Spring的应用上下文。注解@ContextConfiguration会告诉它需要在CDPlayerConfig中加载配置。因为CDPlayerConfig类中包含了@ComponentScan，因此最终的应用上下文中应该包含CompactDiscbean。

为了证明这一点，在测试代码中有一个CompactDisc类型的属性，并且这个属性带有@Autowired注解，以便于将CompactDiscbean注入到测试代码之中（稍后，我会讨论@Autowired）。最后，会有一个简单的测试方法断言cd属性不为null。如果它不为null的话，就意味着Spring能够发现CompactDisc类，自动在Spring上下文中将其创建为bean并将其注入到测试代码之中。

这个代码应该能够通过测试，并以测试成功的颜色显示（在你的测试运行器中，或许会希望出现绿色）。你第一个简单的组件扫描练习就成功了！尽管我们只用它创建了一个bean，但同样是这么少的配置能够用来发现和创建任意数量的bean。在soundsystem包及其子包中，所有带有@Component注解的类都会创建为bean。只添加一行@ComponentScan注解就能自动创建无数个bean，这种权衡还是很划算的。

现在，我们会更加深入地探讨@ComponentScan和@Component，看一下使用组件扫描还能做些什么。

2.2.2 为组件扫描的bean命名

Spring应用上下文中所有的bean都会给定一个ID。在前面的例子中，尽管我们没有明确地为SgtPeppersbean设置ID，但Spring会根据类名为其指定一个ID。具体来讲，这个bean所给定的ID为sgtPeppers，也就是将类名的第一个字母变为小写。

如果想为这个bean设置不同的ID，你所要做的就是将期望的ID作为值传递给@Component注解。比如说，如果想将这个bean标识为lonelyHeartsClub，那么你需要将SgtPeppers类的@Component注解配置为如下所示：

```
@Component("lonelyHeartsClub")
public class SgtPeppers implements CompactDisc {
  ...
}
```

还有另外一种为bean命名的方式，这种方式不使用@Component注解，而是使用Java依赖注入规范（Java Dependency Injection）中所提供的@Named注解来为bean设置ID：

```
package soundsystem;
import javax.inject.Named;

@Named("lonelyHeartsClub")
public class SgtPeppers implements CompactDisc {
  ...
}
```

Spring支持将@Named作为@Component注解的替代方案。两者之间有一些细微的差异，但是在大多数场景中，它们是可以互相替换的。

话虽如此，我更加强烈地喜欢@Component注解，而对于@Named......

怎么说呢，我感觉它的名字起得很不好。它并没有像@Component那样清楚地表明它是做什么的。因此在本书及其示例代码中，我不会再使用@Named。

2.2.3 设置组件扫描的基础包

到现在为止，我们没有为@ComponentScan设置任何属性。这意味着，按照默认规则，它会以配置类所在的包作为基础包（base package）来扫描组件。但是，如果你想扫描不同的包，那该怎么办呢？或者，如果你想扫描多个基础包，那又该怎么办呢？

有一个原因会促使我们明确地设置基础包，那就是我们想要将配置类放在单独的包中，使其与其他的应用代码区分开来。如果是这样的话，那默认的基础包就不能满足要求了。

要满足这样的需求其实也完全没有问题！为了指定不同的基础包，你所需要做的就是在@ComponentScan的value属性中指明包的名称：

```
@Configuration
@ComponentScan("soundsystem")
public class CDPlayerConfig {}
```

如果你想更加清晰地表明你所设置的是基础包，那么你可以通过basePackages属性进行配置：

```
@Configuration
@ComponentScan(basePackages="soundsystem")
public class CDPlayerConfig {}
```

可能你已经注意到了basePackages属性使用的是复数形式。如果你揣测这是不是意味着可以设置多个基础包，那么恭喜你猜对了。如果想要这么做的话，只需要将basePackages属性设置为要扫描包的一个数组即可：

```
@Configuration
@ComponentScan(basePackages={"soundsystem", "video"})
public class CDPlayerConfig {}
```

在上面的例子中，所设置的基础包是以String类型表示的。我认为这是可以的，但这种方法是类型不安全（not type-safe）的。如果你重构代码的话，那么所指定的基础包可能就会出现错误了。

除了将包设置为简单的String类型之外，@ComponentScan还提供了

另外一种方法，那就是将其指定为包中所包含的类或接口：

```
@Configuration
@ComponentScan(basePackageClasses={CDPlayer.class, DVDPlayer.class})
public class CDPlayerConfig {}
```

可以看到，basePackages属性被替换成了basePackageClasses。同时，我们不是再使用String类型的名称来指定包，为basePackageClasses属性所设置的数组中包含了类。这些类所在的包将会作为组件扫描的基础包。

尽管在样例中，我为basePackageClasses设置的是组件类，但是你可以考虑在包中创建一个用来进行扫描的空标记接口（marker interface）。通过标记接口的方式，你依然能够保持对重构友好的接口引用，但是可以避免引用任何实际的应用程序代码（在稍后重构中，这些应用代码有可能会从想要扫描的包中移除掉）。

在你的应用程序中，如果所有的对象都是独立的，彼此之间没有任何依赖，就像SgtPeppersbean这样，那么你所需要的可能就是组件扫描而已。但是，很多对象会依赖其他的对象才能完成任务。这样的话，我们就需要有一种方法能够将组件扫描得到的bean和它们的依赖装配在一起。要完成这项任务，我们需要了解一下Spring自动化配置的另外一方面内容，那就是自动装配。

2.2.4　通过为bean添加注解实现自动装配

简单来说，自动装配就是让Spring自动满足bean依赖的一种方法，在满足依赖的过程中，会在Spring应用上下文中寻找匹配某个bean需求的其他bean。为了声明要进行自动装配，我们可以借助Spring的@Autowired注解。

比方说，考虑程序清单2.6中的CDPlayer类。它的构造器上添加了@Autowired注解，这表明当Spring创建CDPlayerbean的时候，会通过这个构造器来进行实例化并且会传入一个可设置给CompactDisc类型的bean。

程序清单**2.6**　通过自动装配，将一个**CompactDisc**注入到**CDPlayer**之中

```
package soundsystem;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class CDPlayer implements MediaPlayer {
  private CompactDisc cd;

  @Autowired
  public CDPlayer(CompactDisc cd) {
    this.cd = cd;
  }

  public void play() {
    cd.play();
  }

}
```

@Autowired注解不仅能够用在构造器上，还能用在属性的Setter方法上。比如说，如果CDPlayer有一个setCompactDisc()方法，那么可以采用如下的注解形式进行自动装配：

```
@Autowired
public void setCompactDisc(CompactDisc cd) {
  this.cd = cd;
}
```

在Spring初始化bean之后，它会尽可能得去满足bean的依赖，在本例中，依赖是通过带有@Autowired注解的方法进行声明的，也就是setCompactDisc()。

实际上，Setter方法并没有什么特殊之处。@Autowired注解可以用在类的任何方法上。假设CDPlayer类有一个insertDisc()方法，那么@Autowired能够像在setCompactDisc()上那样，发挥完全相同的作用：

```
@Autowired
public void insertDisc(CompactDisc cd) {
  this.cd = cd;
}
```

不管是构造器、Setter方法还是其他的方法，Spring都会尝试满足方法参数上所声明的依赖。假如有且只有一个bean匹配依赖需求的话，那么这个bean将会被装配进来。

如果没有匹配的bean，那么在应用上下文创建的时候，Spring会抛出一个异常。为了避免异常的出现，你可以将@Autowired的required属性设置为false：

```
@Autowired(required=false)
public CDPlayer(CompactDisc cd) {
  this.cd = cd;
}
```

将required属性设置为false时，Spring会尝试执行自动装配，但是如果没有匹配的bean的话，Spring将会让这个bean处于未装配的状态。但是，把required属性设置为false时，你需要谨慎对待。如果在你的代码中没有进行null检查的话，这个处于未装配状态的属性有可能会出现NullPointerException。

如果有多个bean都能满足依赖关系的话，Spring将会抛出一个异常，表明没有明确指定要选择哪个bean进行自动装配。在第3章中，我们会进一步讨论自动装配中的歧义性。

@Autowired是Spring特有的注解。如果你不愿意在代码中到处使用Spring的特定注解来完成自动装配任务的话，那么你可以考虑将其替换为@Inject：

```
package soundsystem;
import javax.inject.Inject;
import javax.inject.Named;

@Named
public class CDPlayer {
  ...

  @Inject
  public CDPlayer(CompactDisc cd) {
    this.cd = cd;
  }

  ...
}
```

@Inject注解来源于Java依赖注入规范，该规范同时还为我们定义了@Named注解。在自动装配中，Spring同时支持@Inject和@Autowired。尽管@Inject和@Autowired之间有着一些细微的差别，但是在大多数场景下，它们都是可以互相替换的。

在@Inject和@Autowired中，我没有特别强烈的偏向性。实际上，在有的项目中，我会发现我同时使用了这两个注解。不过在本书的样例中，我会一直使用@Autowired，而你可以根据自己的情况，选择其中的任意一个。

2.2.5 验证自动装配

现在，我们已经在CDPlayer的构造器中添加了@Autowired注解，Spring将把一个可分配给CompactDisc类型的bean自动注入进来。为了验证这一点，让我们修改一下CDPlayerTest，使其能够借助CDPlayer bean播放CD：

```
package soundsystem;
import static org.junit.Assert.*;
import org.junit.Rule;
import org.junit.Test;
import org.junit.contrib.java.lang.system.StandardOutputStreamLog;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=CDPlayerConfig.class)
public class CDPlayerTest {

  @Rule
  public final StandardOutputStreamLog log =
                        new StandardOutputStreamLog();

  @Autowired
  private MediaPlayer player;

  @Autowired
  private CompactDisc cd;

  @Test
  public void cdShouldNotBeNull() {
    assertNotNull(cd);
  }

  @Test
  public void play() {
    player.play();
    assertEquals(
        "Playing Sgt. Pepper's Lonely Hearts Club Band" +
        " by The Beatles\n",
        log.getLog());
  }

}
```

现在，除了注入CompactDisc，我们还将CDPlayerbean注入到测试代码的player成员变量之中（它是更为通用的MediaPlayer类型）。在play()测试方法中，我们可以调用CDPlayer的play()方法，并断言它的行为与你的预期一致。

在测试代码中使用System.out.println()是稍微有点棘手的事情。因此，该样例中使用了StandardOutputStreamLog，这是来源于System Rules库（http://stefanbirkner.github.io/system-rules/index.html）的一个JUnit规则，该规则能够基于控制台的输出编写断言。在这里，我们断言SgtPeppers.play()方法的输出被发

送到了控制台上。

现在，你已经了解了组件扫描和自动装配的基础知识，在第3章中，当我们介绍如何处理自动装配的歧义性时，还会继续研究组件扫描。

但是现在，我们先将组件扫描和自动装配放在一边，看一下在Spring中如何显式地装配bean，首先从通过Java代码编写配置开始。

2.3 通过Java代码装配bean

尽管在很多场景下通过组件扫描和自动装配实现Spring的自动化配置是更为推荐的方式，但有时候自动化配置的方案行不通，因此需要明确配置Spring。比如说，你想要将第三方库中的组件装配到你的应用中，在这种情况下，是没有办法在它的类上添加`@Component`和`@Autowired`注解的，因此就不能使用自动化装配的方案了。

在这种情况下，你必须要采用显式装配的方式。在进行显式配置的时候，有两种可选方案：Java和XML。在这节中，我们将会学习如何使用Java配置，接下来的一节中将会继续学习Spring的XML配置。

就像我之前所说的，在进行显式配置时，JavaConfig是更好的方案，因为它更为强大、类型安全并且对重构友好。因为它就是Java代码，就像应用程序中的其他Java代码一样。

同时，JavaConfig与其他的Java代码又有所区别，在概念上，它与应用程序中的业务逻辑和领域代码是不同的。尽管它与其他的组件一样都使用相同的语言进行表述，但JavaConfig是配置代码。这意味着它不应该包含任何业务逻辑，JavaConfig也不应该侵入到业务逻辑代码之中。尽管不是必须的，但通常会将JavaConfig放到单独的包中，使它与其他的应用程序逻辑分离开来，这样对于它的意图就不会产生困惑了。

接下来，让我们看一下如何通过JavaConfig显式配置Spring。

2.3.1 创建配置类

在本章前面的程序清单2.3中，我们第一次见识到JavaConfig。让我们重温一下那个样例中的`CDPlayerConfig`：

```
package soundsystem;
import org.springframework.context.annotation.Configuration;

@Configuration
public class CDPlayerConfig {
}
```

创建JavaConfig类的关键在于为其添加@Configuration注解，@Configuration注解表明这个类是一个配置类，该类应该包含在Spring应用上下文中如何创建bean的细节。

到此为止，我们都是依赖组件扫描来发现Spring应该创建的bean。尽管我们可以同时使用组件扫描和显式配置，但是在本节中，我们更加关注于显式配置，因此我将CDPlayerConfig的@ComponentScan注解移除掉了。

移除了@ComponentScan注解，此时的CDPlayerConfig类就没有任何作用了。如果你现在运行CDPlayerTest的话，测试会失败，并且会出现BeanCreation- Exception异常。测试期望被注入CDPlayer和CompactDisc，但是这些bean根本就没有创建，因为组件扫描不会发现它们。

为了再次让测试通过，你可以将@ComponentScan注解添加回去，但是我们这一节关注显式配置，因此让我们看一下如何使用JavaConfig装配CDPlayer和CompactDisc。

2.3.2 声明简单的bean

要在JavaConfig中声明bean，我们需要编写一个方法，这个方法会创建所需类型的实例，然后给这个方法添加@Bean注解。比方说，下面的代码声明了CompactDisc bean：

```
@Bean
public CompactDisc sgtPeppers() {
  return new SgtPeppers();
}
```

@Bean注解会告诉Spring这个方法将会返回一个对象，该对象要注册为Spring应用上下文中的bean。方法体中包含了最终产生bean实例的逻辑。

默认情况下，bean的ID与带有@Bean注解的方法名是一样的。在本例中，bean的名字将会是sgtPeppers。如果你想为其设置成一个不同的名字的话，那么可以重命名该方法，也可以通过name属性指定一

个不同的名字：

```
@Bean(name="lonelyHeartsClubBand")
public CompactDisc sgtPeppers() {
  return new SgtPeppers();
}
```

不管你采用什么方法来为bean命名，bean声明都是非常简单的。方法体返回了一个新的`SgtPeppers`实例。这里是使用Java来进行描述的，因此我们可以发挥Java提供的所有功能，只要最终生成一个`CompactDisc`实例即可。

请稍微发挥一下你的想象力，我们可能希望做一点稍微疯狂的事情，比如说，在一组CD中随机选择一个`CompactDisc`来播放：

```
@Bean
public CompactDisc randomBeatlesCD() {
  int choice = (int) Math.floor(Math.random() * 4);
  if (choice == 0) {
    return new SgtPeppers();
  } else if (choice == 1) {
    return new WhiteAlbum();
  } else if (choice == 2) {
    return new HardDaysNight();
  } else {
    return new Revolver();
  }
}
```

现在，你可以自己想象一下，借助`@Bean`注解方法的形式，我们该如何发挥出Java的全部威力来产生bean。当你想完之后，我们要回过头来看一下在JavaConfig中，如何将`CompactDisc`注入到`CDPlayer`之中。

2.3.3 借助JavaConfig实现注入

我们前面所声明的`CompactDisc` bean是非常简单的，它自身没有其他的依赖。但现在，我们需要声明`CDPlayer`bean，它依赖于`CompactDisc`。在JavaConfig中，要如何将它们装配在一起呢？

在JavaConfig中装配bean的最简单方式就是引用创建bean的方法。例如，下面就是一种声明`CDPlayer`的可行方案：

```
@Bean
public CDPlayer cdPlayer() {
  return new CDPlayer(sgtPeppers());
}
```

cdPlayer()方法像sgtPeppers()方法一样，同样使用了@Bean注解，这表明这个方法会创建一个bean实例并将其注册到Spring应用上下文中。所创建的bean ID为cdPlayer，与方法的名字相同。

cdPlayer()的方法体与sgtPeppers()稍微有些区别。在这里并没有使用默认的构造器构建实例，而是调用了需要传入CompactDisc对象的构造器来创建CDPlayer实例。

看起来，CompactDisc是通过调用sgtPeppers()得到的，但情况并非完全如此。因为sgtPeppers()方法上添加了@Bean注解，Spring将会拦截所有对它的调用，并确保直接返回该方法所创建的bean，而不是每次都对其进行实际的调用。

比如说，假设你引入了一个其他的CDPlayerbean，它和之前的那个bean完全一样：

```
@Bean
public CDPlayer cdPlayer() {
  return new CDPlayer(sgtPeppers());
}

@Bean
public CDPlayer anotherCDPlayer() {
  return new CDPlayer(sgtPeppers());
}
```

假如对sgtPeppers()的调用就像其他的Java方法调用一样的话，那么每个CDPlayer实例都会有一个自己特有的SgtPeppers实例。如果我们讨论的是实际的CD播放器和CD光盘的话，这么做是有意义的。如果你有两台CD播放器，在物理上并没有办法将同一张CD光盘放到两个CD播放器中。

但是，在软件领域中，我们完全可以将同一个SgtPeppers实例注入到任意数量的其他bean之中。默认情况下，Spring中的bean都是单例的，我们并没有必要为第二个CDPlayer bean创建完全相同的SgtPeppers实例。所以，Spring会拦截对sgtPeppers()的调用并确保返回的是Spring所创建的bean，也就是Spring本身在调用sgtPeppers()时所创建的CompactDiscbean。因此，两个CDPlayer bean会得到相同的SgtPeppers实例。

可以看到，通过调用方法来引用bean的方式有点令人困惑。其实还有一种理解起来更为简单的方式：

```
@Bean
public CDPlayer cdPlayer(CompactDisc compactDisc) {
  return new CDPlayer(compactDisc);
}
```

在这里，cdPlayer()方法请求一个CompactDisc作为参数。当Spring调用cdPlayer()创建CDPlayerbean的时候，它会自动装配一个CompactDisc到配置方法之中。然后，方法体就可以按照合适的方式来使用它。借助这种技术，cdPlayer()方法也能够将CompactDisc注入到CDPlayer的构造器中，而且不用明确引用CompactDisc的@Bean方法。

通过这种方式引用其他的bean通常是最佳的选择，因为它不会要求将CompactDisc声明到同一个配置类之中。在这里甚至没有要求CompactDisc必须要在JavaConfig中声明，实际上它可以通过组件扫描功能自动发现或者通过XML来进行配置。你可以将配置分散到多个配置类、XML文件以及自动扫描和装配bean之中，只要功能完整健全即可。不管CompactDisc是采用什么方式创建出来的，Spring都会将其传入到配置方法中，并用来创建CDPlayer bean。

另外，需要提醒的是，我们在这里使用CDPlayer的构造器实现了DI功能，但是我们完全可以采用其他风格的DI配置。比如说，如果你想通过Setter方法注入CompactDisc的话，那么代码看起来应该是这样的：

```
@Bean
public CDPlayer cdPlayer(CompactDisc compactDisc) {
  CDPlayer cdPlayer = new CDPlayer(compactDisc);
  cdPlayer.setCompactDisc(compactDisc);
  return cdPlayer;
}
```

再次强调一遍，带有@Bean注解的方法可以采用任何必要的Java功能来产生bean实例。构造器和Setter方法只是@Bean方法的两个简单样例。这里所存在的可能性仅仅受到Java语言的限制。

2.4 通过XML装配bean

到此为止，我们已经看到了如何让Spring自动发现和装配bean，还看到了如何进行手动干预，即通过JavaConfig显式地装配bean。但是，在装配bean的时候，还有一种可选方案，尽管这种方案可能不太合乎大家的心意，但是它在Spring中已经有很长的历史了。

在Spring刚刚出现的时候，XML是描述配置的主要方式。在Spring的名义下，我们创建了无数行XML代码。在一定程度上，Spring成为了XML配置的同义词。

尽管Spring长期以来确实与XML有着关联，但现在需要明确的是，XML不再是配置Spring的唯一可选方案。Spring现在有了强大的自动化配置和基于Java的配置，XML不应该再是你的第一选择了。

不过，鉴于已经存在那么多基于XML的Spring配置，所以理解如何在Spring中使用XML还是很重要的。但是，我希望本节的内容只是用来帮助你维护已有的XML配置，在完成新的Spring工作时，希望你会使用自动化配置和JavaConfig。

2.4.1 创建XML配置规范

在使用XML为Spring装配bean之前，你需要创建一个新的配置规范。在使用JavaConfig的时候，这意味着要创建一个带有`@Configuration`注解的类，而在XML配置中，这意味着要创建一个XML文件，并且要以`<beans>`元素为根。

最为简单的Spring XML配置如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context">

  <!-- configuration details go here -->

</beans>
```

很容易就能看出来，这个基本的XML配置已经比同等功能的JavaConfig类复杂得多了。作为起步，在JavaConfig中所需要的只是`@Configuration`，但在使用XML时，需要在配置文件的顶部声明多个XML模式（XSD）文件，这些文件定义了配置Spring的XML元素。

> **借助Spring Tool Suite创建XML配置文件**创建和管理Spring XML配置文件的一种简便方式是使用Spring Tool Suite（https://spring.io/tools/sts）。在Spring Tool Suite的菜单中，选择File>New>Spring Bean Configuration File，能够创建Spring XML配置文件，并且可以选择可用的配置命名空间。

用来装配bean的最基本的XML元素包含在spring-beans模式之中，在上面这个XML文件中，它被定义为根命名空间。<beans>是该模式中的一个元素，它是所有Spring配置文件的根元素。

在XML中配置Spring时，还有一些其他的模式。尽管在本书中，我更加关注自动化以及基于Java的配置，但是在本书讲解的过程中，当出现其他模式的时候，我至少会提醒你。

就这样，我们已经有了一个合法的Spring XML配置。不过，它也是一个没有任何用处的配置，因为它（还）没有声明任何bean。为了给予它生命力，让我们重新创建一下CD样例，只不过我们这次使用XML配置，而不是使用JavaConfig和自动化装配。

2.4.2 声明一个简单的<bean>

要在基于XML的Spring配置中声明一个bean，我们要使用spring-beans模式中的另外一个元素：<bean>。<bean>元素类似于JavaConfig中的@Bean注解。我们可以按照如下的方式声明CompactDiscbean：

```
<bean class="soundsystem.SgtPeppers" />
```

这里声明了一个很简单的bean，创建这个bean的类通过class属性来指定的，并且要使用全限定的类名。

因为没有明确给定ID，所以这个bean将会根据全限定类名来进行命名。在本例中，bean的ID将会是“soundsystem.SgtPeppers#0”。其中，“#0”是一个计数的形式，用来区分相同类型的其他bean。如果你声明了另外一个SgtPeppers，并且没有明确进行标识，那么它自动得到的ID将会是“soundsystem.SgtPeppers#1”。

尽管自动化的bean命名方式非常方便，但如果你要稍后引用它的话，那自动产生的名字就没有多大的用处了。因此，通常来讲更好的办法是借助id属性，为每个bean设置一个你自己选择的名字：

```
<bean id="compactDisc" class="soundsystem.SgtPeppers" />
```

稍后将这个bean装配到CDPlayer bean之中的时候，你会用到这个具体的名字。

减少繁琐为了减少XML中繁琐的配置，只对那些需要按名字引用的bean（比如，你需要将对它的引用注入到另外一个bean中）进行明确地命名。

在进一步学习之前，让我们花点时间看一下这个简单bean声明的一些特征。

第一件需要注意的事情就是你不再需要直接负责创建SgtPeppers的实例，在基于JavaConfig的配置中，我们是需要这样做的。当Spring发现这个<bean>元素时，它将会调用SgtPeppers的默认构造器来创建bean。在XML配置中，bean的创建显得更加被动，不过，它并没有JavaConfig那样强大，在JavaConfig配置方式中，你可以通过任何可以想象到的方法来创建bean实例。

另外一个需要注意到的事情就是，在这个简单的<bean>声明中，我们将bean的类型以字符串的形式设置在了class属性中。谁能保证设置给class属性的值是真正的类呢？Spring的XML配置并不能从编译期的类型检查中受益。即便它所引用的是实际的类型，如果你重命名了类，会发生什么呢？

借助**IDE**检查**XML**的合法性使用能够感知Spring功能的IDE，如Spring Tool Suite，能够在很大程度上帮助你确保Spring XML配置的合法性。

以上介绍的只是JavaConfig要优于XML配置的部分原因。我建议在为你的应用选择配置风格时，要记住XML配置的这些缺点。接下来，我们继续Spring XML配置的学习进程，了解如何将SgtPeppersbean注入到CDPlayer之中。

2.4.3 借助构造器注入初始化bean

在Spring XML配置中，只有一种声明bean的方式：使用<bean>元素并指定class属性。Spring会从这里获取必要的信息来创建bean。

但是，在XML中声明DI时，会有多种可选的配置方案和风格。具体到构造器注入，有两种基本的配置方案可供选择：

- <constructor-arg>元素
- 使用Spring 3.0所引入的c-命名空间

两者的区别在很大程度就是是否冗长烦琐。可以看

到，<constructor-arg>元素比使用c-命名空间会更加冗长，从而导致XML更加难以读懂。另外，有些事情<constructor-arg>可以做到，但是使用c-命名空间却无法实现。

在介绍Spring XML的构造器注入时，我们将会分别介绍这两种可选方案。首先，看一下它们各自如何注入bean引用。

构造器注入bean引用

按照现在的定义，CDPlayerbean有一个接受CompactDisc类型的构造器。这样，我们就有了一个很好的场景来学习如何注入bean的引用。

现在已经声明了SgtPeppers bean，并且SgtPeppers类实现了CompactDisc接口，所以实际上我们已经有了一个可以注入到CDPlayerbean中的bean。我们所需要做的就是在XML中声明CDPlayer并通过ID引用SgtPeppers：

```
<bean id="cdPlayer" class="soundsystem.CDPlayer">
  <constructor-arg ref="compactDisc" />
</bean>
```

当Spring遇到这个<bean>元素时，它会创建一个CDPlayer实例。<constructor-arg>元素会告知Spring要将一个ID为compactDisc的bean引用传递到CDPlayer的构造器中。

作为替代的方案，你也可以使用Spring的c-命名空间。c-命名空间是在Spring 3.0中引入的，它是在XML中更为简洁地描述构造器参数的方式。要使用它的话，必须要在XML的顶部声明其模式，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:c="http://www.springframework.org/schema/c"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

 ...

</beans>
```

在c-命名空间和模式声明之后，我们就可以使用它来声明构造器参数了，如下所示：

```
<bean id="cdPlayer" class="soundsystem.CDPlayer"
      c:cd-ref="compactDisc" />
```

在这里，我们使用了c-命名空间来声明构造器参数，它作为<bean>元素的一个属性，不过这个属性的名字有点诡异。图2.1描述了这个属性名是如何组合而成的。

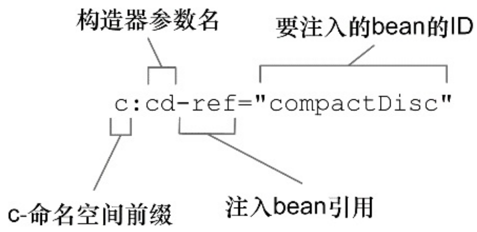

图2.1　通过Spring的c-命名空间将bean引用注入到构造器参数中

属性名以“c:”开头，也就是命名空间的前缀。接下来就是要装配的构造器参数名，在此之后是“-ref”，这是一个命名的约定，它会告诉Spring，正在装配的是一个bean的引用，这个bean的名字是compactDisc，而不是字面量“compactDisc”。

很显然，使用c-命名空间属性要比使用<constructor-arg>元素简练得多。这是我很喜欢它的原因之一。除了更易读之外，当我在编写样例代码时，c-命名空间属性能够更加有助于使代码的长度保持在书的边框之内。

在编写前面的样例时，关于c-命名空间，有一件让我感到困扰的事情就是它直接引用了构造器参数的名称。引用参数的名称看起来有些怪异，因为这需要在编译代码的时候，将调试标志（debug symbol）保存在类代码中。如果你优化构建过程，将调试标志移除掉，那么这种方式可能就无法正常执行了。

替代的方案是我们使用参数在整个参数列表中的位置信息：

```
<bean id="cdPlayer" class="soundsystem.CDPlayer"
      c:_0-ref="compactDisc" />
```

这个c-命名空间属性看起来似乎比上一种方法更加怪异。我将参数的名称替换成了“0”，也就是参数的索引。因为在XML中不允许数字作为属性的第一个字符，因此必须要添加一个下画线作为前缀。

使用索引来识别构造器参数感觉比使用名字更好一些。即便在构建的时候移除掉了调试标志，参数却会依然保持相同的顺序。如果有多个

构造器参数的话，这当然是很有用处的。在这里因为只有一个构造器参数，所以我们还有另外一个方案——根本不用去标示参数：

```
<bean id="cdPlayer" class="soundsystem.CDPlayer"
      c:_-ref="compactDisc" />
```

到目前为止，这是最为奇特的一个c-命名空间属性，这里没有参数索引或参数名。只有一个下画线，然后就是用“-ref”来表明正在装配的是一个引用。

我们已经将引用装配到了其他的bean之中，接下来看一下如何将字面量值（literal value）装配到构造器之中。

将字面量注入到构造器中

迄今为止，我们所做的DI通常指的都是类型的装配——也就是将对象的引用装配到依赖于它们的其他对象之中——而有时候，我们需要做的只是用一个字面量值来配置对象。为了阐述这一点，假设你要创建CompactDisc的一个新实现，如下所示：

```
package soundsystem;

public class BlankDisc implements CompactDisc {

  private String title;
  private String artist;

  public BlankDisc(String title, String artist) {
    this.title = title;
    this.artist = artist;
  }

  public void play() {
    System.out.println("Playing " + title + " by " + artist);
  }

}
```

在SgtPeppers中，唱片名称和艺术家的名字都是硬编码的，但是这个CompactDisc实现与之不同，它更加灵活。像现实中的空磁盘一样，它可以设置成任意你想要的艺术家和唱片名。现在，我们可以将已有的SgtPeppers替换为这个类：

```
<bean id="compactDisc"
      class="soundsystem.BlankDisc">
  <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
  <constructor-arg value="The Beatles" />
</bean>
```

我们再次使用<constructor-arg>元素进行构造器参数的注入。但是这一次我们没有使用“ref”属性来引用其他的bean，而是使用了value属性，通过该属性表明给定的值要以字面量的形式注入到构造器之中。

如果要使用c-命名空间的话，这个例子又该是什么样子呢？第一种方案是引用构造器参数的名字：

```
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      c:_title="Sgt. Pepper's Lonely Hearts Club Band"
      c:_artist="The Beatles" />
```

可以看到，装配字面量与装配引用的区别在于属性名中去掉了“-ref”后缀。与之类似，我们也可以通过参数索引装配相同的字面量值，如下所示：

```
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      c:_0="Sgt. Pepper's Lonely Hearts Club Band"
      c:_1="The Beatles" />
```

XML不允许某个元素的多个属性具有相同的名字。因此，如果有两个或更多的构造器参数的话，我们不能简单地使用下画线进行标示。但是如果只有一个构造器参数的话，我们就可以这样做了。为了完整地展现该功能，假设BlankDisc只有一个构造器参数，这个参数接受唱片的名称。在这种情况下，我们可以在Spring中这样声明它：

```
<bean id="compactDisc" class="soundsystem.BlankDisc"
      c:_="Sgt. Pepper's Lonely Hearts Club Band" />
```

在装配bean引用和字面量值方面，<constructor-arg>和c-命名空间的功能是相同的。但是有一种情况是<constructor-arg>能够实现，c-命名空间却无法做到的。接下来，让我们看一下如何将集合装配到构造器参数中。

装配集合

到现在为止，我们假设CompactDisc在定义时只包含了唱片名称和艺术家的名字。如果现实世界中的CD也是这样的话，那么在技术上就不会任何的进展。CD之所以值得购买是因为它上面所承载的音乐。大多数的CD都会包含十多个磁道，每个磁道上包含一首歌。

如果使用CompactDisc为真正的CD建模，那么它也应该有磁道列表的概念。请考虑下面这个新的BlankDisc：

```
package soundsystem.collections;
import java.util.List;

import soundsystem.CompactDisc;

public class BlankDisc implements CompactDisc {

  private String title;
  private String artist;
  private List<String> tracks;

  public BlankDisc(String title, String artist, List<String> tracks) {
    this.title = title;
    this.artist = artist;
    this.tracks = tracks;
  }

  public void play() {
    System.out.println("Playing " + title + " by " + artist);
    for (String track : tracks) {
      System.out.println("-Track: " + track);
    }
  }

}
```

这个变更会对Spring如何配置bean产生影响，在声明bean的时候，我们必须要提供一个磁道列表。

最简单的办法是将列表设置为null。因为它是一个构造器参数，所以必须要声明它，不过你可以采用如下的方式传递null给它：

```
<bean id="compactDisc" class="soundsystem.BlankDisc">
  <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
  <constructor-arg value="The Beatles" />
  <constructor-arg><null/></constructor-arg>
</bean>
```

<null/>元素所做的事情与你的期望是一样的：将null传递给构造器。这并不是解决问题的好办法，但在注入期它能正常执行。当调用play()方法时，你会遇到NullPointerException异常，因此这并不是理想的方案。

更好的解决方法是提供一个磁道名称的列表。要达到这一点，我们可以有多个可选方案。首先，可以使用<list>元素将其声明为一个列表：

```
<bean id="compactDisc" class="soundsystem.BlankDisc">
  <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
  <constructor-arg value="The Beatles" />
  <constructor-arg>
    <list>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
      <value>With a Little Help from My Friends</value>
      <value>Lucy in the Sky with Diamonds</value>
      <value>Getting Better</value>
      <value>Fixing a Hole</value>
      <!-- ...other tracks omitted for brevity... -->
    </list>
  </constructor-arg>
</bean>
```

其中，`<list>`元素是`<constructor-arg>`的子元素，这表明一个包含值的列表将会传递到构造器中。其中，`<value>`元素用来指定列表中的每个元素。

与之类似，我们也可以使用`<ref>`元素替代`<value>`，实现bean引用列表的装配。例如，假设你有一个`Discography`类，它的构造器如下所示：

```
public Discography(String artist, List<CompactDisc> cds) { ... }
```

那么，你可以采取如下的方式配置`Discography` bean：

```
<bean id="beatlesDiscography"
      class="soundsystem.Discography">
  <constructor-arg value="The Beatles" />
  <constructor-arg>
    <list>
      <ref bean="sgtPeppers" />
      <ref bean="whiteAlbum" />
      <ref bean="hardDaysNight" />
      <ref bean="revolver" />
      ...
    </list>
  </constructor-arg>
</bean>
```

当构造器参数的类型是`java.util.List`时，使用`<list>`元素是合情合理的。尽管如此，我们也可以按照同样的方式使用`<set>`元素：

```xml
<bean id="compactDisc" class="soundsystem.BlankDisc">
  <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
  <constructor-arg value="The Beatles" />
  <constructor-arg>
    <set>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
      <value>With a Little Help from My Friends</value>
      <value>Lucy in the Sky with Diamonds</value>
      <value>Getting Better</value>
      <value>Fixing a Hole</value>
      <!-- ...other tracks omitted for brevity... -->
    </set>
  </constructor-arg>
</bean>
```

<set>和<list>元素的区别不大，其中最重要的不同在于当Spring创建要装配的集合时，所创建的是java.util.Set还是java.util.List。如果是Set的话，所有重复的值都会被忽略掉，存放顺序也不会得以保证。不过无论在哪种情况下，<set>或<list>都可以用来装配List、Set甚至数组。

在装配集合方面，<constructor-arg>比c-命名空间的属性更有优势。目前，使用c-命名空间的属性无法实现装配集合的功能。

使用<constructor-arg>和c-命名空间实现构造器注入时，它们之间还有一些细微的差别。但是到目前为止，我们所涵盖的内容已经足够了，尤其是像我之前所建议的那样，要首选基于Java的配置而不是XML。因此，与其不厌其烦地花费时间讲述如何使用XML进行构造器注入，还不如看一下如何使用XML来装配属性。

2.4.4 设置属性

到目前为止，CDPlayer和BlankDisc类完全是通过构造器注入的，没有使用属性的Setter方法。接下来，我们就看一下如何使用Spring XML实现属性注入。假设属性注入的CDPlayer如下所示：

```java
package soundsystem;
import org.springframework.beans.factory.annotation.Autowired;
import soundsystem.CompactDisc;
import soundsystem.MediaPlayer;

public class CDPlayer implements MediaPlayer {
  private CompactDisc compactDisc;

  @Autowired
  public void setCompactDisc(CompactDisc compactDisc) {
    this.compactDisc = compactDisc;
  }

  public void play() {
    compactDisc.play();
  }
}
```

该选择构造器注入还是属性注入呢？作为一个通用的规则，我倾向于对强依赖使用构造器注入，而对可选性的依赖使用属性注入。按照这个规则，我们可以说对于BlankDisc来讲，唱片名称、艺术家以及磁道列表是强依赖，因此构造器注入是正确的方案。不过，对于CDPlayer来讲，它对CompactDisc是强依赖还是可选性依赖可能会有些争议。虽然我不太认同，但你可能会觉得即便没有将CompactDisc装入进去，CDPlayer依然还能具备一些有限的功能。

现在，CDPlayer没有任何的构造器（除了隐含的默认构造器），它也没有任何的强依赖。因此，你可以采用如下的方式将其声明为Spring bean：

```xml
<bean id="cdPlayer"
      class="soundsystem.CDPlayer" />
```

Spring在创建bean的时候不会有任何的问题，但是CDPlayerTest会因为出现NullPointerException而导致测试失败，因为我们并没有注入CDPlayer的compactDisc属性。不过，按照如下的方式修改XML，就能解决该问题：

```xml
<bean id="cdPlayer"
      class="soundsystem.CDPlayer">
  <property name="compactDisc" ref="compactDisc" />
</bean>
```

<property>元素为属性的Setter方法所提供的功能与<constructor-arg>元素为构造器所提供的功能是一样的。在本例中，它引用了ID为compactDisc的bean（通过ref属性），并将其注入到compactDisc属性中（通过setCompactDisc()方

法）。如果你现在运行测试的话，它应该就能通过了。

我们已经知道，Spring为`<constructor-arg>`元素提供了c-命名空间作为替代方案，与之类似，Spring提供了更加简洁的p-命名空间，作为`<property>`元素的替代方案。为了启用p-命名空间，必须要在XML文件中与其他的命名空间一起对其进行声明：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
  ...
</bean>
```

我们可以使用p-命名空间，按照以下的方式装配compactDisc属性：

```
<bean id="cdPlayer"
      class="soundsystem.CDPlayer"
      p:compactDisc-ref="compactDisc" />
```

p-命名空间中属性所遵循的命名约定与c-命名空间中的属性类似。图2.2阐述了p-命名空间属性是如何组成的。

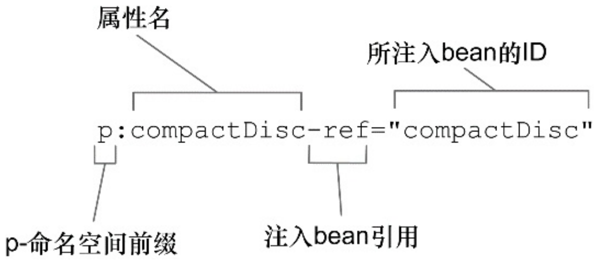

图2.2　借助Spring的p-命名空间，将bean引用注入到属性中

首先，属性的名字使用了“`p:`”前缀，表明我们所设置的是一个属性。接下来就是要注入的属性名。最后，属性的名称以“`-ref`”结尾，这会提示Spring要进行装配的是引用，而不是字面量。

将字面量注入到属性中

属性也可以注入字面量，这与构造器参数非常类似。作为示例，我们重新看一下`BlankDisc` bean。不过，`BlankDisc`这次完全通过属性注入进行配置，而不是构造器注入。新的`BlankDisc`类如下所示：

```java
package soundsystem;
import java.util.List;
import soundsystem.CompactDisc;

public class BlankDisc implements CompactDisc {

  private String title;
  private String artist;
  private List<String> tracks;

  public void setTitle(String title) {
    this.title = title;
  }

  public void setArtist(String artist) {
    this.artist = artist;
  }
  public void setTracks(List<String> tracks) {
    this.tracks = tracks;
  }

  public void play() {
    System.out.println("Playing " + title + " by " + artist);
    for (String track : tracks) {
      System.out.println("-Track: " + track);
    }
  }

}
```

现在，它不再强制要求我们装配任何的属性。你可以按照如下的方式创建一个`BlankDisc`bean，它的所有属性全都是空的：

```xml
<bean id="reallyBlankDisc"
      class="soundsystem.BlankDisc" />
```

当然，如果在装配bean的时候不设置这些属性，那么在运行期CD播放器将不能正常播放内容。`play()`方法可能会遇到的输出内容是“Playing null by null”，随之会抛出`NullPointerException`异常，这是因为我们没有指定任何的磁道。所以，我们需要装配这些属性，可以借助`<property>`元素的`value`属性实现该功能：

```
<bean id="compactDisc"
      class="soundsystem.BlankDisc">
  <property name="title"
               value="Sgt. Pepper's Lonely Hearts Club Band" />
  <property name="artist" value="The Beatles" />
  <property name="tracks">
    <list>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
      <value>With a Little Help from My Friends</value>
      <value>Lucy in the Sky with Diamonds</value>
      <value>Getting Better</value>
      <value>Fixing a Hole</value>
      <!-- ...other tracks omitted for brevity... -->
    </list>
  </property>
</bean>
```

在这里，除了使用<property>元素的value属性来设置title和artist，我们还使用了内嵌的<list>元素来设置tracks属性，这与之前通过<constructor-arg>装配tracks是完全一样的。

另外一种可选方案就是使用p-命名空间的属性来完成该功能：

```
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      p:title="Sgt. Pepper's Lonely Hearts Club Band"
      p:artist="The Beatles">
  <property name="tracks">
    <list>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
     <value>With a Little Help from My Friends</value>
     <value>Lucy in the Sky with Diamonds</value>
     <value>Getting Better</value>
     <value>Fixing a Hole</value>
     <!-- ...other tracks omitted for brevity... -->
   </list>
  </property>
</bean>
```

与c-命名空间一样，装配bean引用与装配字面量的唯一区别在于是否带有“-ref”后缀。如果没有“-ref”后缀的话，所装配的就是字面量。

但需要注意的是，我们不能使用p-命名空间来装配集合，没有便利的方式使用p-命名空间来指定一个值（或bean引用）的列表。但是，我们可以使用Spring util-命名空间中的一些功能来简化BlankDiscbean。

首先，需要在XML中声明util-命名空间及其模式：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util.xsd">

    ...

</beans>
```

util-命名空间所提供的功能之一就是<util:list>元素，它会创建一个列表的bean。借助<util:list>，我们可以将磁道列表转移到BlankDisc bean之外，并将其声明到单独的bean之中，如下所示：

```xml
<util:list id="trackList">
  <value>Sgt. Pepper's Lonely Hearts Club Band</value>
  <value>With a Little Help from My Friends</value>
  <value>Lucy in the Sky with Diamonds</value>
  <value>Getting Better</value>
  <value>Fixing a Hole</value>
  <!-- ...other tracks omitted for brevity... -->
</util:list>
```

现在，我们能够像使用其他的bean那样，将磁道列表bean注入到BlankDisc bean的tracks属性中：

```xml
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      p:title="Sgt. Pepper's Lonely Hearts Club Band"
      p:artist="The Beatles"
      p:tracks-ref="trackList" />
```

<util:list>只是util-命名空间中的多个元素之一。表2.1列出了util-命名空间提供的所有元素。

在需要的时候，你可能会用到util-命名空间中的部分成员。但现在，在结束本章前，我们看一下如何将自动化配置、JavaConfig以及XML配置混合并匹配在一起。

表**2.1　Spring util-**命名空间中的元素

元 素	描 述

<util:constant>	引用某个类型的public static域，并将其暴露为bean
util:list	创建一个java.util.List类型的bean，其中包含值或引用
util:map	创建一个java.util.Map类型的bean，其中包含值或引用
util:properties	创建一个java.util.Properties类型的bean
util:property-path	引用一个bean的属性（或内嵌属性），并将其暴露为bean
util:set	创建一个java.util.Set类型的bean，其中包含值或引用

2.5 导入和混合配置

在典型的Spring应用中，我们可能会同时使用自动化和显式配置。即便你更喜欢通过JavaConfig实现显式配置，但有的时候XML却是最佳的方案。

幸好在Spring中，这些配置方案都不是互斥的。你尽可以将JavaConfig的组件扫描和自动装配和/或XML配置混合在一起。实际上，就像在2.2.1小节中所看到的，我们至少需要有一点显式配置来启用组件扫描和自动装配。

关于混合配置，第一件需要了解的事情就是在自动装配时，它并不在意要装配的bean来自哪里。自动装配的时候会考虑到Spring容器中所有的bean，不管它是在JavaConfig或XML中声明的还是通过组件扫描获取到的。

你可能会想在显式配置时，比如在XML配置和Java配置中该如何引用bean呢。让我们先看一下如何在JavaConfig中引用XML配置的bean。

2.5.1 在JavaConfig中引用XML配置

现在，我们临时假设`CDPlayerConfig`已经变得有些笨重，我们想要将其进行拆分。当然，它目前只定义了两个bean，远远称不上复杂的Spring配置。不过，我们假设两个bean就已经太多了。

我们所能实现的一种方案就是将BlankDisc从CDPlayerConfig拆分出来，定义到它自己的CDConfig类中，如下所示：

```
package soundsystem;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class CDConfig {
  @Bean
  public CompactDisc compactDisc() {
    return new SgtPeppers();
  }
}
```

compactDisc()方法已经从CDPlayerConfig中移除掉了，我们需要有一种方式将这两个类组合在一起。一种方法就是在CDPlayerConfig中使用@Import注解导入CDConfig：

```
package soundsystem;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import(CDConfig.class)
public class CDPlayerConfig {

  @Bean
  public CDPlayer cdPlayer(CompactDisc compactDisc) {
    return new CDPlayer(compactDisc);
  }

}
```

或者采用一个更好的办法，也就是不在CDPlayerConfig中使用@Import，而是创建一个更高级别的SoundSystemConfig，在这个类中使用@Import将两个配置类组合在一起：

```
package soundsystem;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import({CDPlayerConfig.class, CDConfig.class})
public class SoundSystemConfig {
}
```

不管采用哪种方式，我们都将CDPlayer的配置与BlankDisc的配置分开了。现在，我们假设（基于某些原因）希望通过XML来配置BlankDisc，如下所示：

```
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      c:_0="Sgt. Pepper's Lonely Hearts Club Band"
      c:_1="The Beatles">
  <constructor-arg>
    <list>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
      <value>With a Little Help from My Friends</value>
      <value>Lucy in the Sky with Diamonds</value>
      <value>Getting Better</value>
      <value>Fixing a Hole</value>
      <!-- ...other tracks omitted for brevity... -->
    </list>
  </constructor-arg>
</bean>
```

现在BlankDisc配置在了XML之中，我们该如何让Spring同时加载它和其他基于Java的配置呢？

答案是@ImportResource注解，假设BlankDisc定义在名为cd-config.xml的文件中，该文件位于根类路径下，那么可以修改SoundSystemConfig，让它使用@ImportResource注解，如下所示：

```
package soundsystem;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.ImportResource;

@Configuration
@Import(CDPlayerConfig.class)
@ImportResource("classpath:cd-config.xml")
public class SoundSystemConfig {
}
```

两个bean——配置在JavaConfig中的CDPlayer以及配置在XML中BlankDisc——都会被加载到Spring容器之中。因为CDPlayer中带有@Bean注解的方法接受一个CompactDisc作为参数，因此BlankDisc将会装配进来，此时与它是通过XML配置的没有任何关系。

让我们继续这个练习，但是这一次，我们需要在XML中引用JavaConfig声明的bean。

2.5.2 在XML配置中引用JavaConfig

假设你正在使用Spring基于XML的配置并且你已经意识到XML逐渐变得无法控制。像前面一样，我们正在处理的是两个bean，但事情实际上会变得更加糟糕。在被无数的尖括号淹没之前，我们决定将XML配

置文件进行拆分。

在JavaConfig配置中，我们已经展现了如何使用@Import和@ImportResource来拆分JavaConfig类。在XML中，我们可以使用import元素来拆分XML配置。

比如，假设希望将BlankDisc bean拆分到自己的配置文件中，该文件名为cd-config.xml，这与我们之前使用@ImportResource是一样的。我们可以在XML配置文件中使用<import>元素来引用该文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:c="http://www.springframework.org/schema/c"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean class="soundsystem.CDConfig" />

  <bean id="cdPlayer"
        class="soundsystem.CDPlayer"
        c:cd-ref="compactDisc" />

</beans>
```

现在，我们假设不再将BlankDisc配置在XML之中，而是将其配置在JavaConfig中，CDPlayer则继续配置在XML中。基于XML的配置该如何引用一个JavaConfig类呢？

事实上，答案并不那么直观。<import>元素只能导入其他的XML配置文件，并没有XML元素能够导入JavaConfig类。

但是，有一个你已经熟知的元素能够用来将Java配置导入到XML配置中：<bean>元素。为了将JavaConfig类导入到XML配置中，我们可以这样声明bean：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:c="http://www.springframework.org/schema/c"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean class="soundsystem.CDConfig" />

  <bean id="cdPlayer"
        class="soundsystem.CDPlayer"
        c:cd-ref="compactDisc" />

</beans>
```

采用这样的方式，两种配置——其中一个使用XML描述，另一个使用Java描述——被组合在了一起。类似地，你可能还希望创建一个更高层次的配置文件，这个文件不声明任何的bean，只是负责将两个或更多的配置组合起来。例如，你可以将`CDConfig` bean从之前的XML文件中移除掉，而是使用第三个配置文件将这两个组合在一起：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:c="http://www.springframework.org/schema/c"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean class="soundsystem.CDConfig" />

  <import resource="cdplayer-config.xml" />

</beans>
```

不管使用JavaConfig还是使用XML进行装配，我通常都会创建一个根配置（root configuration），也就是这里展现的这样，这个配置会将两个或更多的装配类和/或XML文件组合起来。我也会在根配置中启用组件扫描（通过`<context:component-scan>`或`@ComponentScan`）。你会在本书的很多例子中看到这种技术。

2.6 小结

Spring框架的核心是Spring容器。容器负责管理应用中组件的生命周期，它会创建这些组件并保证它们的依赖能够得到满足，这样的话，组件才能完成预定的任务。

在本章中，我们看到了在Spring中装配bean的三种主要方式：自动化配置、基于Java的显式配置以及基于XML的显式配置。不管你采用什么方式，这些技术都描述了Spring应用中的组件以及这些组件之间的关系。

我同时建议尽可能使用自动化配置，以避免显式配置所带来的维护成本。但是，如果你确实需要显式配置Spring的话，应该优先选择基于Java的配置，它比基于XML的配置更加强大、类型安全并且易于重构。在本书中的例子中，当决定如何装配组件时，我都会遵循这样的指导意见。

因为依赖注入是Spring中非常重要的组成部分，所以本章中介绍的技术在本书中所有的地方都会用到。基于这些基础知识，下一章将会介

绍一些更为高级的bean装配技术，这些技术能够让你更加充分地发挥Spring容器的威力。

第3章 高级装配

本章内容：

- Spring profile
- 条件化的bean声明
- 自动装配与歧义性
- bean的作用域
- Spring表达式语言

在上一章中，我们看到了一些最为核心的bean装配技术。你可能会发现上一章学到的知识有很大的用处。但是，bean装配所涉及的领域并不仅仅局限于上一章 所学习到的内容。Spring提供了多种技巧，借助它们可以实现更为高级的bean装配功能。

在本章中，我们将会深入介绍一些这样的高级技术。本章中所介绍的技术也许你不会天天都用到，但这并不意味着它们的价值会因此而降低。

3.1 环境与profile

在开发软件的时候，有一个很大的挑战就是将应用程序从一个环境迁移到另外一个环境。开发阶段中，某些环境相关做法可能并不适合迁移到生产环境中，甚至即便迁移过去也无法正常工作。数据库配置、加密算法以及与外部系统的集成是跨环境部署时会发生变化的几个典型例子。

比如，考虑一下数据库配置。在开发环境中，我们可能会使用嵌入式数据库，并预先加载测试数据。例如，在Spring配置类中，我们可能会在一个带有`@Bean`注解的方法上使用`EmbeddedDatabaseBuilder`：

```
@Bean(destroyMethod="shutdown")
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .addScript("classpath:schema.sql")
        .addScript("classpath:test-data.sql")
        .build();
}
```

这会创建一个类型为javax.sql.DataSource的bean，这个bean是如何创建出来的才是最有意思的。使用EmbeddedDatabaseBuilder会搭建一个嵌入式的Hypersonic数据库，它的模式（schema）定义在schema.sql中，测试数据则是通过test-data.sql加载的。

当你在开发环境中运行集成测试或者启动应用进行手动测试的时候，这个DataSource是很有用的。每次启动它的时候，都能让数据库处于一个给定的状态。

尽管EmbeddedDatabaseBuilder创建的DataSource非常适于开发环境，但是对于生产环境来说，这会是一个糟糕的选择。在生产环境的配置中，你可能会希望使用JNDI从容器中获取一个DataSource。在这样场景中，如下的@Bean方法会更加合适：

```
@Bean
public DataSource dataSource() {
  JndiObjectFactoryBean jndiObjectFactoryBean =
      new JndiObjectFactoryBean();
  jndiObjectFactoryBean.setJndiName("jdbc/myDS");
  jndiObjectFactoryBean.setResourceRef(true);
  jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
  return (DataSource) jndiObjectFactoryBean.getObject();
}
```

通过JNDI获取DataSource能够让容器决定该如何创建这个DataSource，甚至包括切换为容器管理的连接池。即便如此，JNDI管理的DataSource更加适合于生产环境，对于简单的集成和开发测试环境来说，这会带来不必要的复杂性。

同时，在QA环境中，你可以选择完全不同的DataSource配置，可以配置为Commons DBCP连接池，如下所示：

```
@Bean(destroyMethod="close")
public DataSource dataSource() {
  BasicDataSource dataSource = new BasicDataSource();
  dataSource.setUrl("jdbc:h2:tcp://dbserver/~/test");
  dataSource.setDriverClassName("org.h2.Driver");
  dataSource.setUsername("sa");
  dataSource.setPassword("password");
  dataSource.setInitialSize(20);
  dataSource.setMaxActive(30);

  return dataSource;
}
```

显然，这里展现的三个版本的dataSource()方法互不相同。虽然它们都会生成一个类型为javax.sql.DataSource的bean，但它们的相似点也仅限于此了。每个方法都使用了完全不同的策略来生

成DataSource bean。

再次强调的是，这里的讨论并不是如何配置DataSource（我们将会在第10章更详细地讨论这个话题）。看起来很简单的DataSource实际上并不是那么简单。这是一个很好的例子，它表现了在不同的环境中某个bean会有所不同。我们必须要有一种方法来配置DataSource，使其在每种环境下都会选择最为合适的配置。

其中一种方式就是在单独的配置类（或XML文件）中配置每个bean，然后在构建阶段（可能会使用Maven的profiles）确定要将哪一个配置编译到可部署的应用中。这种方式的问题在于要为每种环境重新构建应用。当从开发阶段迁移到QA阶段时，重新构建也许算不上什么大问题。但是，从QA阶段迁移到生产阶段时，重新构建可能会引入bug并且会在QA团队的成员中带来不安的情绪。

值得庆幸的是，Spring所提供的解决方案并不需要重新构建。

3.1.1 配置profile bean

Spring为环境相关的bean所提供的解决方案其实与构建时的方案没有太大的差别。当然，在这个过程中需要根据环境决定该创建哪个bean和不创建哪个bean。不过Spring并不是在构建的时候做出这样的决策，而是等到运行时再来确定。这样的结果就是同一个部署单元（可能会是WAR文件）能够适用于所有的环境，没有必要进行重新构建。

在3.1版本中，Spring引入了bean profile的功能。要使用profile，你首先要将所有不同的bean定义整理到一个或多个profile之中，在将应用部署到每个环境时，要确保对应的profile处于激活（active）的状态。

在Java配置中，可以使用@Profile注解指定某个bean属于哪一个profile。例如，在配置类中，嵌入式数据库的DataSource可能会配置成如下所示：

```
package com.myapp;
import javax.activation.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration
@Profile("dev")
public class DevelopmentProfileConfig {

  @Bean(destroyMethod="shutdown")
  public DataSource dataSource() {
      return new EmbeddedDatabaseBuilder()
          .setType(EmbeddedDatabaseType.H2)
          .addScript("classpath:schema.sql")
          .addScript("classpath:test-data.sql")
          .build();
  }

}
```

我希望你能够注意的是`@Profile`注解应用在了类级别上。它会告诉Spring这个配置类中的bean只有在`dev` profile激活时才会创建。如果`dev` profile没有激活的话，那么带有`@Bean`注解的方法都会被忽略掉。

同时，你可能还需要有一个适用于生产环境的配置，如下所示：

```
package com.myapp;
import javax.activation.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration
@Profile("prod")
public class ProductionProfileConfig {

  @Bean
  public DataSource dataSource() {
    JndiObjectFactoryBean jndiObjectFactoryBean =
        new JndiObjectFactoryBean();
    jndiObjectFactoryBean.setJndiName("jdbc/myDS");
    jndiObjectFactoryBean.setResourceRef(true);
    jndiObjectFactoryBean.setProxyInterface(
        javax.sql.DataSource.class);
    return (DataSource) jndiObjectFactoryBean.getObject();
  }

}
```

在本例中，只有`prod` profile激活的时候，才会创建对应的bean。

在Spring 3.1中，只能在类级别上使用`@Profile`注解。不过，从Spring 3.2开始，你也可以在方法级别上使用`@Profile`注解，与`@Bean`注解一同使用。这样的话，就能将这两个bean的声明放到同一个配置类之中，如下所示：

程序清单**3.1　@Profile**注解基于激活的**profile**实现**bean**的装配

```
package com.myapp;
import javax.activation.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration
public class DataSourceConfig {

  @Bean(destroyMethod="shutdown")
  @Profile("dev")                    <---- 为 dev profile 装配的 bean
  public DataSource embeddedDataSource() {
      return new EmbeddedDatabaseBuilder()
          .setType(EmbeddedDatabaseType.H2)
          .addScript("classpath:schema.sql")
          .addScript("classpath:test-data.sql")
          .build();
  }

  @Bean
  @Profile("prod")                     <---- 为 prod profile 装配的 bean
  public DataSource jndiDataSource() {
    JndiObjectFactoryBean jndiObjectFactoryBean =
        new JndiObjectFactoryBean();
    jndiObjectFactoryBean.setJndiName("jdbc/myDS");
    jndiObjectFactoryBean.setResourceRef(true);
    jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
    return (DataSource) jndiObjectFactoryBean.getObject();
  }

}
```

这里有个问题需要注意，尽管每个`DataSource` bean都被声明在一个profile中，并且只有当规定的profile激活时，相应的bean才会被创建，但是可能会有其他的bean并没有声明在一个给定的profile范围内。没有指定profile的bean始终都会被创建，与激活哪个profile没有关系。

在XML中配置profile

我们也可以通过`<beans>`元素的`profile`属性，在XML中配置profile bean。例如，为了在XML中定义适用于开发阶段的嵌入式数据

库DataSourcebean，我们可以创建如下所示的XML文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd"
   profile="dev">

  <jdbc:embedded-database id="dataSource">
    <jdbc:script location="classpath:schema.sql" />
    <jdbc:script location="classpath:test-data.sql" />
  </jdbc:embedded-database>

</beans>
```

与之类似，我们也可以将profile设置为prod，创建适用于生产环境的从JNDI获取的DataSource bean。同样，可以创建基于连接池定义的DataSource bean，将其放在另外一个XML文件中，并标注为qaprofile。所有的配置文件都会放到部署单元之中（如WAR文件），但是只有profile属性与当前激活profile相匹配的配置文件才会被用到。

你还可以在根<beans>元素中嵌套定义<beans>元素，而不是为每个环境都创建一个profile XML文件。这能够将所有的profile bean定义放到同一个XML文件中，如下所示：

程序清单**3.2**　重复使用元素来指定多个**profile**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:p="http://www.springframework.org/schema/p"
  xsi:schemaLocation="
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <beans profile="dev">                          <-- dev profile 的 bean
    <jdbc:embedded-database id="dataSource">
      <jdbc:script location="classpath:schema.sql" />
      <jdbc:script location="classpath:test-data.sql" />
    </jdbc:embedded-database>
  </beans>

  <beans profile="qa">                           <-- qa profile 的 bean
    <bean id="dataSource"
          class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close"
          p:url="jdbc:h2:tcp://dbserver/~/test"
          p:driverClassName="org.h2.Driver"
          p:username="sa"
          p:password="password"
          p:initialSize="20"
          p:maxActive="30" />
  </beans>

  <beans profile="prod">                         <-- prod profile 的 bean
     <jee:jndi-lookup id="dataSource"
                      jndi-name="jdbc/myDatabase"
                      resource-ref="true"
                      proxy-interface="javax.sql.DataSource" />
  </beans>
</beans>
```

除了所有的bean定义到了同一个XML文件之中，这种配置方式与定义在单独的XML文件中的实际效果是一样的。这里有三个bean，类型都是`javax.sql.DataSource`，并且ID都是`dataSource`。但是在运行时，只会创建一个bean，这取决于处于激活状态的是哪个profile。

那么问题来了：我们该怎样激活某个profile呢？

3.1.2 激活profile

Spring在确定哪个profile处于激活状态时，需要依赖两个独立的属性：`spring.profiles.active`和`spring.profiles.default`。如果设置了`spring.profiles.active`属性的话，那么它的值就会用来确定哪个profile是激活的。但如果没有设置`spring.profiles.active`

属性的话，那Spring将会查找spring.profiles.default的值。如果spring.profiles.active和spring.profiles.default均没有设置的话，那就没有激活的profile，因此只会创建那些没有定义在profile中的bean。

有多种方式来设置这两个属性：

- 作为DispatcherServlet的初始化参数；
- 作为Web应用的上下文参数；
- 作为JNDI条目；
- 作为环境变量；
- 作为JVM的系统属性；
- 在集成测试类上，使用@ActiveProfiles注解设置。

你尽可以选择spring.profiles.active和spring.profiles.default的最佳组合方式以满足需求，我将这样的自主权留给读者。

我所喜欢的一种方式是使用DispatcherServlet的参数将spring.profiles.default设置为开发环境的profile，我会在Servlet上下文中进行设置（为了兼顾到ContextLoaderListener）。例如，在Web应用中，设置spring.profiles.default的web.xml文件会如下所示：

程序清单**3.3**　在**Web**应用的**web.xml**文件中设置默认的**profile**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>

  <context-param>
    <param-name>spring.profiles.default</param-name>      <-- 为上下文设置
    <param-value>dev</param-value>                            默认的 profile
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
      <param-name>spring.profiles.default</param-name>    <-- 为 Servlet 设置
      <param-value>dev</param-value>                          默认的 profile
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

按照这种方式设置spring.profiles.default，所有的开发人员都能从版本控制软件中获得应用程序源码，并使用开发环境的设置（如嵌入式数据库）运行代码，而不需要任何额外的配置。

当应用程序部署到QA、生产或其他环境之中时，负责部署的人根据情况使用系统属性、环境变量或JNDI设置spring.profiles.active即可。当设置spring.profiles.active以后，至于spring.profiles.default置成什么值就已经无所谓了；系统会优先使用spring.profiles.active中所设置的profile。

你可能已经注意到了，在spring.profiles.active和spring.profiles.default中，profile使用的都是复数形式。这意味着你可以同时激活多个profile，这可以通过列出多个profile名称，并以逗号分隔来实现。当然，同时启用dev和prod profile可能也没有太大的意义，不过你可以同时设置多个彼此不相关的profile。

使用profile进行测试

当运行集成测试时，通常会希望采用与生产环境（或者是生产环境的部分子集）相同的配置进行测试。但是，如果配置中的bean定义在了profile中，那么在运行测试时，我们就需要有一种方式来启用合适的profile。

Spring提供了`@ActiveProfiles`注解，我们可以使用它来指定运行测试时要激活哪个profile。在集成测试时，通常想要激活的是开发环境的profile。例如，下面的测试类片段展现了使用`@ActiveProfiles`激活`dev` profile：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={PersistenceTestConfig.class})
@ActiveProfiles("dev")
public class PersistenceTest {
  ...
}
```

在条件化创建bean方面，Spring的profile机制是一种很棒的方法，这里的条件要基于哪个profile处于激活状态来判断。Spring 4.0中提供了一种更为通用的机制来实现条件化的bean定义，在这种机制之中，条件完全由你来确定。让我们看一下如何使用Spring 4和`@Conditional`注解定义条件化的bean。

3.2　条件化的bean

假设你希望一个或多个bean只有在应用的类路径下包含特定的库时才创建。或者我们希望某个bean只有当另外某个特定的bean也声明了之后才会创建。我们还可能要求只有某个特定的环境变量设置之后，才会创建某个bean。

在Spring 4之前，很难实现这种级别的条件化配置，但是Spring 4引入了一个新的`@Conditional`注解，它可以用到带有`@Bean`注解的方法上。如果给定的条件计算结果为`true`，就会创建这个bean，否则的话，这个bean会被忽略。

例如，假设有一个名为`MagicBean`的类，我们希望只有设置了`magic`环境属性的时候，Spring才会实例化这个类。如果环境中没有这个属性，那么`MagicBean`将会被忽略。在程序清单3.4所展现的配置中，使用`@Conditional`注解条件化地配置了`MagicBean`。

程序清单**3.4**　条件化地配置**bean**

```
@Bean
@Conditional(MagicExistsCondition.class)        <—— 条件化地创建 bean
public MagicBean magicBean() {
  return new MagicBean();
}
```

可以看到，`@Conditional`中给定了一个`Class`，它指明了条件——在本例中，也就是`MagicExistsCondition`。`@Conditional`将会通过`Condition`接口进行条件对比：

```
public interface Condition {
    boolean matches(ConditionContext ctxt,
                    AnnotatedTypeMetadata metadata);
}
```

设置给`@Conditional`的类可以是任意实现了`Condition`接口的类型。可以看出来，这个接口实现起来很简单直接，只需提供`matches()`方法的实现即可。如果`matches()`方法返回`true`，那么就会创建带有`@Conditional`注解的bean。如果`matches()`方法返回`false`，将不会创建这些bean。

在本例中，我们需要创建`Condition`的实现并根据环境中是否存在`magic`属性来做出决策。程序清单3.5展现了`MagicExistsCondition`，这是完成该功能的`Condition`实现类：

程序清单**3.5**　在**Condition**中检查是否存在**magic**属性

```
package com.habuma.restfun;
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;
import org.springframework.util.ClassUtils;

public class MagicExistsCondition implements Condition {

  public boolean matches(
          ConditionContext context, AnnotatedTypeMetadata metadata) {
    Environment env = context.getEnvironment();
    return env.containsProperty("magic");        <—— 检查 magic 属性
  }

}
```

在上面的程序清单中，`matches()`方法很简单但功能强大。它通过给定的`ConditionContext`对象进而得到`Environment`对象，并使用这个对象检查环境中是否存在名为`magic`的环境属性。在本例中，属性的值是什么无所谓，只要属性存在即可满足要求。如果满足

这个条件的话，matches()方法就会返回true。所带来的结果就是条件能够得到满足，所有@Conditional注解上引用MagicExistsCondition的bean都会被创建。

话说回来，如果这个属性不存在的话，就无法满足条件，matches()方法会返回false，这些bean都不会被创建。

MagicExistsCondition中只是使用了ConditionContext得到的Environment，但Condition实现的考量因素可能会比这更多。matches()方法会得到ConditionContext和AnnotatedTypeMetadata对象用来做出决策。

ConditionContext是一个接口，大致如下所示：

```
public interface ConditionContext {
  BeanDefinitionRegistry getRegistry();
  ConfigurableListableBeanFactory getBeanFactory();
  Environment getEnvironment();
  ResourceLoader getResourceLoader();
  ClassLoader getClassLoader();
}
```

通过ConditionContext，我们可以做到如下几点：

- 借助getRegistry()返回的BeanDefinitionRegistry检查bean定义；
- 借助getBeanFactory()返回的ConfigurableListableBeanFactory检查bean是否存在，甚至探查bean的属性；
- 借助getEnvironment()返回的Environment检查环境变量是否存在以及它的值是什么；
- 读取并探查getResourceLoader()返回的ResourceLoader所加载的资源；
- 借助getClassLoader()返回的ClassLoader加载并检查类是否存在。

AnnotatedTypeMetadata则能够让我们检查带有@Bean注解的方法上还有什么其他的注解。像ConditionContext一样，AnnotatedTypeMetadata也是一个接口。它如下所示：

```
public interface AnnotatedTypeMetadata {
  boolean isAnnotated(String annotationType);
  Map<String, Object> getAnnotationAttributes(String annotationType);
  Map<String, Object> getAnnotationAttributes(
          String annotationType, boolean classValuesAsString);
  MultiValueMap<String, Object> getAllAnnotationAttributes(
          String annotationType);
  MultiValueMap<String, Object> getAllAnnotationAttributes(
          String annotationType, boolean classValuesAsString);
}
```

借助`isAnnotated()`方法，我们能够判断带有`@Bean`注解的方法是不是还有其他特定的注解。借助其他的那些方法，我们能够检查`@Bean`注解的方法上其他注解的属性。

非常有意思的是，从Spring 4开始，`@Profile`注解进行了重构，使其基于`@Conditional`和`Condition`实现。作为如何使用`@Conditional`和`Condition`的例子，我们来看一下在Spring 4中，`@Profile`是如何实现的。

`@Profile`注解如下所示：

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Documented
@Conditional(ProfileCondition.class)
public @interface Profile {
  String[] value();
}
```

> 注意：@Profile本身也使用了@Conditional注解，并且引用ProfileCondition作为Condition实现。如下所示，ProfileCondition实现了Condition接口，并且在做出决策的过程中，考虑到了ConditionContext和AnnotatedTypeMetadata中的多个因素。

程序清单**3.6**　**ProfileCondition**检查某个**bean profile**是否可用

```
class ProfileCondition implements Condition {
  public boolean matches(
        ConditionContext context, AnnotatedTypeMetadata metadata) {
    if (context.getEnvironment() != null) {
      MultiValueMap<String, Object> attrs =
          metadata.getAllAnnotationAttributes(Profile.class.getName());
      if (attrs != null) {
        for (Object value : attrs.get("value")) {
          if (context.getEnvironment()
                  .acceptsProfiles(((String[]) value))) {
            return true;
          }
        }
        return false;
      }
    }
    return true;
  }
}
```

我们可以看到，ProfileCondition通过AnnotatedTypeMetadata得到了用于@Profile注解的所有属性。借助该信息，它会明确地检查value属性，该属性包含了bean的profile名称。然后，它根据通过ConditionContext得到的Environment来检查［借助acceptsProfiles()方法］该profile是否处于激活状态。

3.3 处理自动装配的歧义性

在第2章中，我们已经看到如何使用自动装配让Spring完全负责将bean引用注入到构造参数和属性中。自动装配能够提供很大的帮助，因为它会减少装配应用程序组件时所需要的显式配置的数量。

不过，仅有一个bean匹配所需的结果时，自动装配才是有效的。如果不仅有一个bean能够匹配结果的话，这种歧义性会阻碍Spring自动装配属性、构造器参数或方法参数。

为了阐述自动装配的歧义性，假设我们使用@Autowired注解标注了setDessert()方法：

```
@Autowired
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

在本例中，Dessert是一个接口，并且有三个类实现了这个接口，分别为Cake、Cookies和IceCream：

```
@Component
public class Cake implements Dessert { ... }

@Component
public class Cookies implements Dessert { ... }
@Component
public class IceCream implements Dessert { ... }
```

因为这三个实现均使用了@Component注解，在组件扫描的时候，能够发现它们并将其创建为Spring应用上下文里面的bean。然后，当Spring试图自动装配setDessert()中的Dessert参数时，它并没有唯一、无歧义的可选值。在从多种甜点中做出选择时，尽管大多数人并不会有什么困难，但是Spring却无法做出选择。Spring此时别无他法，只好宣告失败并抛出异常。更精确地讲，Spring会抛出NoUniqueBeanDefinitionException：

```
nested exception is
  org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [com.desserteater.Dessert] is defined:
  expected single matching bean but found 3: cake,cookies,iceCream
```

当然，使用吃甜点的样例来阐述自动装配在遇到歧义性时所面临的问题多少有些牵强。在实际中，自动装配歧义性的问题其实比你想象中的更为罕见。就算这种歧义性确实是个问题，但更常见的情况是给定的类型只有一个实现类，因此自动装配能够很好地运行。

但是，当确实发生歧义性的时候，Spring提供了多种可选方案来解决这样的问题。你可以将可选bean中的某一个设为首选（primary）的bean，或者使用限定符（qualifier）来帮助Spring将可选的bean的范围缩小到只有一个bean。

3.3.1　标示首选的bean

如果你像我一样，喜欢所有类型的甜点，如蛋糕、饼干、冰激凌……它们都很美味。但如果只能在其中选择一种甜点的话，那你最喜欢的是哪一种呢？

在声明bean的时候，通过将其中一个可选的bean设置为首选（primary）bean能够避免自动装配时的歧义性。当遇到歧义性的时候，Spring将会使用首选的bean，而不是其他可选的bean。实际上，你所声明就是“最喜欢”的bean。

假设冰激凌就是你最喜欢的甜点。在Spring中，可以通过`@Primary`来表达最喜欢的方案。`@Primary`能够与`@Component`组合用在组件扫描的bean上，也可以与`@Bean`组合用在Java配置的bean声明中。比如，下面的代码展现了如何将`@Component`注解的`IceCream` bean声明为首选的bean：

```
@Component
@Primary
public class IceCream implements Dessert { ... }
```

或者，如果你通过Java配置显式地声明`IceCream`，那么`@Bean`方法应该如下所示：

```
@Bean
@Primary
public Dessert iceCream() {
    return new IceCream();
}
```

如果你使用XML配置bean的话，同样可以实现这样的功能。<bean>元素有一个primary属性用来指定首选的bean：

```
<bean id="iceCream"
      class="com.desserteater.IceCream"
      primary="true" />
```

不管你采用什么方式来标示首选bean，效果都是一样的，都是告诉Spring在遇到歧义性的时候要选择首选的bean。

但是，如果你标示了两个或更多的首选bean，那么它就无法正常工作了。比如，假设Cake类如下所示：

```
@Component
@Primary
public class Cake implements Dessert { ... }
```

现在，有两个首选的Dessert bean：Cake和IceCream。这带来了新的歧义性问题。就像Spring无法从多个可选的bean中做出选择一样，它也无法从多个首选的bean中做出选择。显然，如果不止一个bean被设置成了首选bean，那实际上也就是没有首选bean了。

就解决歧义性问题而言，限定符是一种更为强大的机制，下面就将对其进行介绍。

3.3.2　限定自动装配的bean

设置首选bean的局限性在于@Primary无法将可选方案的范围限定到唯一一个无歧义性的选项中。它只能标示一个优先的可选方案。当首选bean的数量超过一个时，我们并没有其他的方法进一步缩小可选范围。

与之相反，Spring的限定符能够在所有可选的bean上进行缩小范围的操作，最终能够达到只有一个bean满足所规定的限制条件。如果将所有的限定符都用上后依然存在歧义性，那么你可以继续使用更多的限定符来缩小选择范围。

@Qualifier注解是使用限定符的主要方式。它可以与@Autowired和@Inject协同使用，在注入的时候指定想要注入进去的是哪个bean。例如，我们想要确保要将IceCream注入到setDessert()之中：

```
@Autowired
@Qualifier("iceCream")
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

这是使用限定符的最简单的例子。为`@Qualifier`注解所设置的参数就是想要注入的bean的ID。所有使用`@Component`注解声明的类都会创建为bean，并且bean的ID为首字母变为小写的类名。因此，`@Qualifier("iceCream")`指向的是组件扫描时所创建的bean，并且这个bean是`IceCream`类的实例。

实际上，还有一点需要补充一下。更准确地讲，`@Qualifier("iceCream")`所引用的bean要具有String类型的“iceCream”作为限定符。如果没有指定其他的限定符的话，所有的bean都会给定一个默认的限定符，这个限定符与bean的ID相同。因此，框架会将具有“iceCream”限定符的bean注入到`setDessert()`方法中。这恰巧就是ID为`iceCream`的bean，它是`IceCream`类在组件扫描的时候创建的。

基于默认的bean ID作为限定符是非常简单的，但这有可能会引入一些问题。如果你重构了`IceCream`类，将其重命名为Gelato的话，那此时会发生什么情况呢？如果这样的话，bean的ID和默认的限定符会变为`gelato`，这就无法匹配`setDessert()`方法中的限定符。自动装配会失败。

这里的问题在于`setDessert()`方法上所指定的限定符与要注入的bean的名称是紧耦合的。对类名称的任意改动都会导致限定符失效。

创建自定义的限定符

我们可以为bean设置自己的限定符，而不是依赖于将bean ID作为限定符。在这里所需要做的就是在bean声明上添加`@Qualifier`注解。例如，它可以与`@Component`组合使用，如下所示：

```
@Component
@Qualifier("cold")
public class IceCream implements Dessert { ... }
```

在这种情况下，cold限定符分配给了`IceCream`bean。因为它没有耦合类名，因此你可以随意重构`IceCream`的类名，而不必担心会破坏自动装配。在注入的地方，只要引用cold限定符就可以了：

```
@Autowired
@Qualifier("cold")
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

值得一提的是，当通过Java配置显式定义bean的时候，@Qualifier也可以与@Bean注解一起使用：

```
@Bean
@Qualifier("cold")
public Dessert iceCream() {
    return new IceCream();
}
```

当使用自定义的@Qualifier值时，最佳实践是为bean选择特征性或描述性的术语，而不是使用随意的名字。在本例中，我将IceCream bean描述为“cold”bean。在注入的时候，可以将这个需求理解为“给我一个凉的甜点”，这其实就是描述的IceCream。类似地，我可以将Cake描述为“soft”，将Cookie描述为“crispy”。

使用自定义的限定符注解

面向特性的限定符要比基于bean ID的限定符更好一些。但是，如果多个bean都具备相同特性的话，这种做法也会出现问题。例如，如果引入了这个新的Dessert bean，会发生什么情况呢：

```
@Component
@Qualifier("cold")
public class Popsicle implements Dessert { ... }
```

不会吧？！现在我们有了两个带有“cold”限定符的甜点。在自动装配Dessert bean的时候，我们再次遇到了歧义性的问题，需要使用更多的限定符来将可选范围限定到只有一个bean。

可能想到的解决方案就是在注入点和bean定义的地方同时再添加另外一个@Qualifier注解。IceCream类大致就会如下所示：

```
@Component
@Qualifier("cold")
@Qualifier("creamy")
public class IceCream implements Dessert { ... }
```

Popsicle类同样也可能再添加另外一个@Qualifier注解：

```
@Component
@Qualifier("cold")
@Qualifier("fruity")
public class Popsicle implements Dessert { ... }
```

在注入点中，我们可能会使用这样的方式来将范围缩小到IceCream：

```
@Autowired
@Qualifier("cold")
@Qualifier("creamy")
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

这里只有一个小问题：Java不允许在同一个条目上重复出现相同类型的多个注解。[1]如果你试图这样做的话，编译器会提示错误。在这里，使用@Qualifier注解并没有办法（至少没有直接的办法）将自动装配的可选bean缩小范围至仅有一个可选的bean。

但是，我们可以创建自定义的限定符注解，借助这样的注解来表达bean所希望限定的特性。这里所需要做的就是创建一个注解，它本身要使用@Qualifier注解来标注。这样我们将不再使用@Qualifier("cold")，而是使用自定义的@Cold注解，该注解的定义如下所示：

```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,
         ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Cold { }
```

同样，你可以创建一个新的@Creamy注解来代替@Qualifier("creamy")：

```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,
         ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Creamy { }
```

当你不想用@Qualifier注解的时候，可以类似地创建@Soft、@Crispy和@Fruity。通过在定义时添加@Qualifier注解，它们就具有了@Qualifier注解的特性。它们本身实际上就成为了限定符注解。

现在，我们可以重新看一下IceCream，并为其添加@Cold和@Creamy注解，如下所示：

```
@Component
@Cold
@Creamy
public class IceCream implements Dessert { ... }
```

类似地，Popsicle类可以添加@Cold和@Fruity注解：

```
@Component
@Cold
@Fruity
public class Popsicle implements Dessert { ... }
```

最终，在注入点，我们使用必要的限定符注解进行任意组合，从而将可选范围缩小到只有一个bean满足需求。为了得到IceCream bean，setDessert()方法可以这样使用注解：

```
@Autowired
@Cold
@Creamy
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

通过声明自定义的限定符注解，我们可以同时使用多个限定符，不会再有Java编译器的限制或错误。与此同时，相对于使用原始的@Qualifier并借助String类型来指定限定符，自定义的注解也更为类型安全。

让我们近距离观察一下setDessert()方法以及它的注解，这里并没有在任何地方明确指定要将IceCream自动装配到该方法中。相反，我们使用所需bean的特性来进行指定，即@Cold和@Creamy。因此，setDessert()方法依然能够与特定的Dessert实现保持解耦。任意满足这些特征的bean都是可以的。在当前选择Dessert实现时，恰好如此，IceCream是唯一能够与之匹配的bean。

在本节和前面的节中，我们讨论了几种通过自定义注解扩展Spring的方式。为了创建自定义的条件化注解，我们创建一个新的注解并在这个注解上添加了@Conditional。为了创建自定义的限定符注解，我们创建一个新的注解并在这个注解上添加了@Qualifier。这种技术可以用到很多的Spring注解中，从而能够将它们组合在一起形成特定目标的自定义注解。

现在我们来看一下如何在不同的作用域中声明bean。

3.4 bean的作用域

在默认情况下，Spring应用上下文中所有bean都是作为以单例（singleton）的形式创建的。也就是说，不管给定的一个bean被注入

到其他bean多少次，每次所注入的都是同一个实例。

在大多数情况下，单例bean是很理想的方案。初始化和垃圾回收对象实例所带来的成本只留给一些小规模任务，在这些任务中，让对象保持无状态并且在应用中反复重用这些对象可能并不合理。

有时候，可能会发现，你所使用的类是易变的（mutable），它们会保持一些状态，因此重用是不安全的。在这种情况下，将class声明为单例的bean就不是什么好主意了，因为对象会被污染，稍后重用的时候会出现意想不到的问题。

Spring定义了多种作用域，可以基于这些作用域创建bean，包括：

- 单例（Singleton）：在整个应用中，只创建bean的一个实例。
- 原型（Prototype）：每次注入或者通过Spring应用上下文获取的时候，都会创建一个新的bean实例。
- 会话（Session）：在Web应用中，为每个会话创建一个bean实例。
- 请求（Rquest）：在Web应用中，为每个请求创建一个bean实例。

单例是默认的作用域，但是正如之前所述，对于易变的类型，这并不合适。如果选择其他的作用域，要使用`@Scope`注解，它可以与`@Component`或`@Bean`一起使用。

例如，如果你使用组件扫描来发现和声明bean，那么你可以在bean的类上使用`@Scope`注解，将其声明为原型bean：

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class Notepad { ... }
```

这里，使用`ConfigurableBeanFactory`类的`SCOPE_PROTOTYPE`常量设置了原型作用域。你当然也可以使用`@Scope("prototype")`，但是使用`SCOPE_PROTOTYPE`常量更加安全并且不易出错。

如果你想在Java配置中将`Notepad`声明为原型bean，那么可以组合使用`@Scope`和`@Bean`来指定所需的作用域：

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public Notepad notepad() {
  return new Notepad();
}
```

同样，如果你使用XML来配置bean的话，可以使用`<bean>`元素的scope属性来设置作用域：

```
<bean id="notepad"
      class="com.myapp.Notepad"
      scope="prototype" />
```

不管你使用哪种方式来声明原型作用域，每次注入或从Spring应用上下文中检索该bean的时候，都会创建新的实例。这样所导致的结果就是每次操作都能得到自己的`Notepad`实例。

3.4.1 使用会话和请求作用域

在Web应用中，如果能够实例化在会话和请求范围内共享的bean，那将是非常有价值的事情。例如，在典型的电子商务应用中，可能会有一个bean代表用户的购物车。如果购物车是单例的话，那么将会导致所有的用户都会向同一个购物车中添加商品。另一方面，如果购物车是原型作用域的，那么在应用中某一个地方往购物车中添加商品，在应用的另外一个地方可能就不可用了，因为在这里注入的是另外一个原型作用域的购物车。

就购物车bean来说，会话作用域是最为合适的，因为它与给定的用户关联性最大。要指定会话作用域，我们可以使用`@Scope`注解，它的使用方式与指定原型作用域是相同的：

```
@Component
@Scope(
    value=WebApplicationContext.SCOPE_SESSION,
    proxyMode=ScopedProxyMode.INTERFACES)
public ShoppingCart cart() { ... }
```

这里，我们将`value`设置成了`WebApplicationContext`中的`SCOPE_SESSION`常量（它的值是`session`）。这会告诉Spring为Web应用中的每个会话创建一个`ShoppingCart`。这会创建多个`ShoppingCart` bean的实例，但是对于给定的会话只会创建一个实例，在当前会话相关的操作中，这个bean实际上相当于单例的。

要注意的是，`@Scope`同时还有一个`proxyMode`属性，它被设置成了`ScopedProxyMode.INTERFACES`。这个属性解决了将会话或请求

作用域的bean注入到单例bean中所遇到的问题。在描述proxyMode属性之前，我们先来看一下proxyMode所解决问题的场景。

假设我们要将ShoppingCart bean注入到单例StoreService bean的Setter方法中，如下所示：

```
@Component
public class StoreService {

  @Autowired
  public void setShoppingCart(ShoppingCart shoppingCart) {
    this.shoppingCart = shoppingCart;
  }

  ...
}
```

因为StoreService是一个单例的bean，会在Spring应用上下文加载的时候创建。当它创建的时候，Spring会试图将ShoppingCart bean注入到setShoppingCart()方法中。但是ShoppingCart bean是会话作用域的，此时并不存在。直到某个用户进入系统，创建了会话之后，才会出现ShoppingCart实例。

另外，系统中将会有多个ShoppingCart实例：每个用户一个。我们并不想让Spring注入某个固定的ShoppingCart实例到StoreService中。我们希望的是当StoreService处理购物车功能时，它所使用的ShoppingCart实例恰好是当前会话所对应的那一个。

Spring并不会将实际的ShoppingCart bean注入到StoreService中，Spring会注入一个到ShoppingCart bean的代理，如图3.1所示。这个代理会暴露与ShoppingCart相同的方法，所以StoreService会认为它就是一个购物车。但是，当StoreService调用ShoppingCart的方法时，代理会对其进行懒解析并将调用委托给会话作用域内真正的ShoppingCart bean。

现在，我们带着对这个作用域的理解，讨论一下proxyMode属性。如配置所示，proxyMode属性被设置成了ScopedProxyMode.INTERFACES，这表明这个代理要实现ShoppingCart接口，并将调用委托给实现bean。

如果ShoppingCart是接口而不是类的话，这是可以的（也是最为理想的代理模式）。但如果ShoppingCart是一个具体的类的话，Spring就没有办法创建基于接口的代理了。此时，它必须使用CGLib来生成基于类的代理。所以，如果bean类型是具体类的话，我们必须

要将proxyMode属性设置为ScopedProxyMode.TARGET_CLASS，以此来表明要以生成目标类扩展的方式创建代理。

尽管我主要关注了会话作用域，但是请求作用域的bean会面临相同的装配问题。因此，请求作用域的bean应该也以作用域代理的方式进行注入。

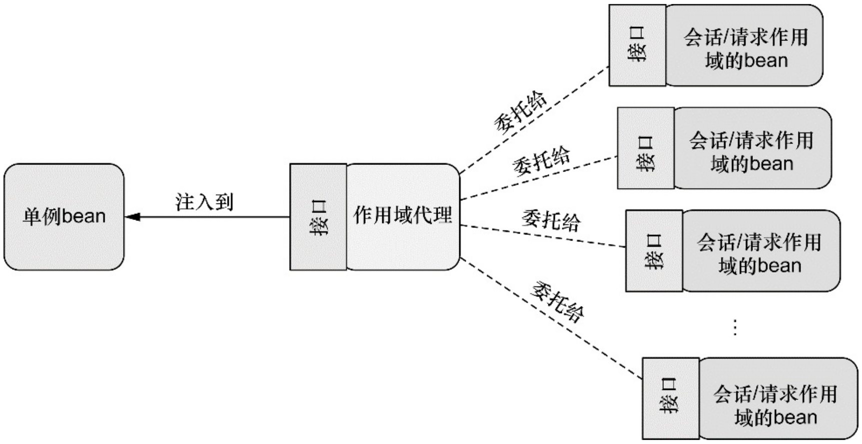

图3.1　作用域代理能够延迟注入请求和会话作用域的bean

3.4.2　在XML中声明作用域代理

如果你需要使用XML来声明会话或请求作用域的bean，那么就不能使用@Scope注解及其proxyMode属性了。<bean>元素的scope属性能够设置bean的作用域，但是该怎样指定代理模式呢？

要设置代理模式，我们需要使用Spring aop命名空间的一个新元素：

```
<bean id="cart"
      class="com.myapp.ShoppingCart"
      scope="session">
  <aop:scoped-proxy />
</bean>
```

<aop:scoped-proxy>是与@Scope注解的proxyMode属性功能相同的Spring XML配置元素。它会告诉Spring为bean创建一个作用域代理。默认情况下，它会使用CGLib创建目标类的代理。但是我们也可以将proxy-target-class属性设置为false，进而要求它生成基于接口的代理：

```
<bean id="cart"
      class="com.myapp.ShoppingCart"
      scope="session">
  <aop:scoped-proxy proxy-target-class="false" />
</bean>
```

为了使用<aop:scoped-proxy>元素，我们必须在XML配置中声明Spring的aop命名空间：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

...

</beans>
```

在第4章中，当我们使用Spring和面向切面编程的时候，会讨论Spring aop命名空间的更多知识。不过，在结束本章的内容之前，我们来看一下Spring高级配置的另外一个可选方案：Spring表达式语言（Spring Expression Language）。

3.5 运行时值注入

当讨论依赖注入的时候，我们通常所讨论的是将一个bean引用注入到另一个bean的属性或构造器参数中。它通常来讲指的是将一个对象与另一个对象进行关联。

但是bean装配的另外一个方面指的是将一个值注入到bean的属性或者构造器参数中。我们在第2章中已经进行了很多值装配，如将专辑的名字装配到BlankDisc bean的构造器或title属性中。例如，我们可能按照这样的方式来组装BlankDisc：

```
@Bean
public CompactDisc sgtPeppers() {
  return new BlankDisc(
      "Sgt. Pepper's Lonely Hearts Club Band",
      "The Beatles");
}
```

尽管这实现了你的需求，也就是为BlankDisc bean设置title和artist，但它在实现的时候是将值硬编码在配置类中的。与之类似，如果使用XML的话，那么值也会是硬编码的：

```
<bean id="sgtPeppers"
      class="soundsystem.BlankDisc"
      c:_title="Sgt. Pepper's Lonely Hearts Club Band"
      c:_artist="The Beatles" />
```

有时候硬编码是可以的，但有的时候，我们可能会希望避免硬编码值，而是想让这些值在运行时再确定。为了实现这些功能，Spring提供了两种在运行时求值的方式：

- 属性占位符（Property placeholder）。
- Spring表达式语言（SpEL）。

很快你就会发现这两种技术的用法是类似的，不过它们的目的和行为是有所差别的。让我们先看一下属性占位符，在这两者中它较为简单，然后再看一下更为强大的SpEL。

3.5.1 注入外部的值

在Spring中，处理外部值的最简单方式就是声明属性源并通过Spring的`Environment`来检索属性。例如，程序清单3.7展现了一个基本的Spring配置类，它使用外部的属性来装配`BlankDisc` bean。

程序清单**3.7** 使用**@PropertySource**注解和**Environment**

```
package com.soundsystem;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;

@Configuration
@PropertySource("classpath:/com/soundsystem/app.properties")    <-- 声明属性源
 public class ExpressiveConfig {

  @Autowired
  Environment env;

  @Bean
 public BlankDisc disc() {
   return new BlankDisc(
       env.getProperty("disc.title"),          <-- 检索属性值
        env.getProperty("disc.artist"));
 }
```

在本例中，`@PropertySource`引用了类路径中一个名为app.properties的文件。它大致会如下所示：

```
disc.title=Sgt. Peppers Lonely Hearts Club Band
disc.artist=The Beatles
```

这个属性文件会加载到Spring的Environment中，稍后可以从这里检索属性。同时，在disc()方法中，会创建一个新的BlankDisc，它的构造器参数是从属性文件中获取的，而这是通过调用getProperty()实现的。

深入学习Spring的Environment

当我们去了解Environment的时候会发现，程序清单3.7所示的getProperty()方法并不是获取属性值的唯一方法，getProperty()方法有四个重载的变种形式：

- `String getProperty(String key)`
- `String getProperty(String key, String defaultValue)`
- `T getProperty(String key, Class<T> type)`
- `T getProperty(String key, Class<T> type, T defaultValue)`

前两种形式的getProperty()方法都会返回String类型的值。我们已经在程序清单3.7中看到了如何使用第一种getProperty()方法。但是，你可以稍微对@Bean方法进行一下修改，这样在指定属性不存在的时候，会使用一个默认值：

```
@Bean
public BlankDisc disc() {
return new BlankDisc(
    env.getProperty("disc.title", "Rattle and Hum"),
    env.getProperty("disc.artist", "U2"));
}
```

剩下的两种getProperty()方法与前面的两种非常类似，但是它们不会将所有的值都视为String类型。例如，假设你想要获取的值所代表的含义是连接池中所维持的连接数量。如果我们从属性文件中得到的是一个String类型的值，那么在使用之前还需要将其转换为Integer类型。但是，如果使用重载形式的getProperty()的话，就能非常便利地解决这个问题：

```
int connectionCount =
    env.getProperty("db.connection.count", Integer.class, 30);
```

Environment还提供了几个与属性相关的方法，如果你在使用getProperty()方法的时候没有指定默认值，并且这个属性没有定义的话，获取到的值是null。如果你希望这个属性必须要定义，那么可以使用getRequiredProperty()方法，如下所示：

```
@Bean
public BlankDisc disc() {
return new BlankDisc(
    env.getRequiredProperty("disc.title"),
    env.getRequiredProperty("disc.artist"));
}
```

在这里，如果disc.title或disc.artist属性没有定义的话，将会抛出IllegalStateException异常。

如果想检查一下某个属性是否存在的话，那么可以调用Environment的containsProperty()方法：

```
boolean titleExists = env.containsProperty("disc.title");
```

最后，如果想将属性解析为类的话，可以使用getPropertyAsClass()方法：

```
Class<CompactDisc> cdClass =
    env.getPropertyAsClass("disc.class", CompactDisc.class);
```

除了属性相关的功能以外，Environment还提供了一些方法来检查哪些profile处于激活状态：

- String[] getActiveProfiles()：返回激活profile名称的数组；
- String[] getDefaultProfiles()：返回默认profile名称的数组；
- boolean acceptsProfiles(String... profiles)：如果environment支持给定profile的话，就返回true。

在程序清单3.6中，我们已经看到了如何使用acceptsProfiles()。在那个例子中，Environment是从ConditionContext中获取到的，在bean创建之前，使用acceptsProfiles()方法来确保给定bean所需的profile处于激活状态。通常来讲，我们并不会频繁使用Environment相关的方法，但是知道有这些方法还是有好处的。

直接从Environment中检索属性是非常方便的，尤其是在Java配置中装配bean的时候。但是，Spring也提供了通过占位符装配属性的方法，这些占位符的值会来源于一个属性源。

解析属性占位符

Spring一直支持将属性定义到外部的属性的文件中，并使用占位符值将其插入到Spring bean中。在Spring装配中，占位符的形式为使用“${ ... }”包装的属性名称。作为样例，我们可以在XML中按照如下的方式解析BlankDisc构造器参数：

```
<bean id="sgtPeppers"
      class="soundsystem.BlankDisc"
      c:_title="${disc.title}"
      c:_artist="${disc.artist}" />
```

可以看到，title构造器参数所给定的值是从一个属性中解析得到的，这个属性的名称为disc.title。artist参数装配的是名为disc.artist的属性值。按照这种方式，XML配置没有使用任何硬编码的值，它的值是从配置文件以外的一个源中解析得到的。（我们稍后会讨论这些属性是如何解析的。）

如果我们依赖于组件扫描和自动装配来创建和初始化应用组件的话，那么就没有指定占位符的配置文件或类了。在这种情况下，我们可以使用@Value注解，它的使用方式与@Autowired注解非常相似。比如，在BlankDisc类中，构造器可以改成如下所示：

```
public BlankDisc(
      @Value("${disc.title}") String title,
      @Value("${disc.artist}") String artist) {
  this.title = title;
  this.artist = artist;
}
```

为了使用占位符，我们必须要配置一个PropertyPlaceholderConfigurer bean或PropertySourcesPlaceholderConfigurer bean。从Spring 3.1开始，推荐使用PropertySourcesPlaceholderConfigurer，因为它能够基于Spring Environment及其属性源来解析占位符。

如下的@Bean方法在Java中配置了PropertySourcesPlaceholderConfigurer：

```
@Bean
public
static PropertySourcesPlaceholderConfigurer placeholderConfigurer() {
  return new PropertySourcesPlaceholderConfigurer();
}
```

如果你想使用XML配置的话，Spring context命名空间中的<context:propertyplaceholder>元素将会为你生成PropertySourcesPlaceholderConfigurer bean：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

  <context:property-placeholder />

</beans>
```

解析外部属性能够将值的处理推迟到运行时，但是它的关注点在于根据名称解析来自于Spring `Environment`和属性源的属性。而Spring表达式语言提供了一种更通用的方式在运行时计算所要注入的值。

3.5.2 使用Spring表达式语言进行装配

Spring 3引入了Spring表达式语言（Spring Expression Language，SpEL），它能够以一种强大和简洁的方式将值装配到bean属性和构造器参数中，在这个过程中所使用的表达式会在运行时计算得到值。使用SpEL，你可以实现超乎想象的装配效果，这是使用其他的装配技术难以做到的（甚至是不可能的）。

SpEL拥有很多特性，包括：

- 使用bean的ID来引用bean;
- 调用方法和访问对象的属性;
- 对值进行算术、关系和逻辑运算;
- 正则表达式匹配;
- 集合操作。

在本书后面的内容中你可以看到，SpEL能够用在依赖注入以外的其他地方。例如，Spring Security支持使用SpEL表达式定义安全限制规则。另外，如果你在Spring MVC应用中使用Thymeleaf模板作为视图的话，那么这些模板可以使用SpEL表达式引用模型数据。

作为起步，我们看几个SpEL表达式的样例，以及如何将其注入到bean中。然后我们会深入学习一些SpEL的基础表达式，它们能够组合起来形成更为强大的表达式。

SpEL样例

SpEL是一种非常灵活的表达式语言，所以在本书中不可能面面俱到地介绍它的各种用法。但是我们可以展示几个基本的例子，这些例子会激发你的灵感，有助于你编写自己的表达式。

需要了解的第一件事情就是SpEL表达式要放到“#{ ... }”之中，这与属性占位符有些类似，属性占位符需要放到“${ ... }”之中。下面所展现的可能是最简单的SpEL表达式了：

```
#{1}
```

除去“#{ ... }”标记之后，剩下的就是SpEL表达式体了，也就是一个数字常量。这个表达式的计算结果就是数字1，这恐怕并不会让你感到丝毫惊讶。

当然，在实际的应用程序中，我们可能并不会使用这么简单的表达式。我们可能会使用更加有意思的表达式，如：

```
#{T(System).currentTimeMillis()}
```

它的最终结果是计算表达式的那一刻当前时间的毫秒数。`T()`表达式会将`java.lang.System`视为Java中对应的类型，因此可以调用其`static`修饰的`currentTimeMillis()`方法。

SpEL表达式也可以引用其他的bean或其他bean的属性。例如，如下的表达式会计算得到ID为`sgtPeppers`的bean的`artist`属性：

```
#{sgtPeppers.artist}
```

我们还可以通过`systemProperties`对象引用系统属性：

```
#{systemProperties['disc.title']}
```

这只是SpEL的几个基础样例。在本章结束之前，你还会看到很多这样的表达式。但是，在此之前，让我们看一下在bean装配的时候如何使用这些表达式。

如果通过组件扫描创建bean的话，在注入属性和构造器参数时，我们可以使用`@Value`注解，这与之前看到的属性占位符非常类似。不过，在这里我们所使用的不是占位符表达式，而是SpEL表达式。例如，下面的样例展现了`BlankDisc`，它会从系统属性中获取专辑名称和艺术家的名字：

```
public BlankDisc(
      @Value("#{systemProperties['disc.title']}") String title,
      @Value("#{systemProperties['disc.artist']}") String artist) {
  this.title = title;
  this.artist = artist;
}
```

在XML配置中，你可以将SpEL表达式传入`<property>`或`<constructor-arg>`的`value`属性中，或者将其作为p-命名空间或c-命名空间条目的值。例如，在如下`BlankDisc` bean的XML声明中，构造器参数就是通过SpEL表达式设置的：

```
<bean id="sgtPeppers"
      class="soundsystem.BlankDisc"
      c:_title="#{systemProperties['disc.title']}"
      c:_artist="#{systemProperties['disc.artist']}" />
```

我们已经看过了几个简单的样例，也学习了如何将SpEL解析得到的值注入到bean中，那现在就来继续学习一下SpEL所支持的基础表达式吧。

表示字面值

我们在前面已经看到了一个使用SpEL来表示整数字面量的样例。它实际上还可以用来表示浮点数、`String`值以及Boolean值。

下面的SpEL表达式样例所表示的就是浮点值：

```
#{3.14159}
```

数值还可以使用科学记数法的方式进行表示。如下面的表达式计算得到的值就是98,700：

```
#{9.87E4}
```

SpEL表达式也可以用来计算`String`类型的字面值，如：

```
#{'Hello'}
```

最后，字面值`true`和`false`的计算结果就是它们对应的Boolean类型的值。例如：

```
#{false}
```

在SpEL中使用字面值其实没有太大的意思，毕竟将整型属性设置为1，或者将Boolean属性设置为`false`时，我们并不需要使用SpEL。我

承认在SpEL表达式中，只包含字面值情况并没有太大的用处。但需要记住的一点是，更有意思的SpEL表达式是由更简单的表达式组成的，因此了解在SpEL中如何使用字面量还是很有用处的。当组合更为复杂的表达式时，你迟早会用到它们。

引用**bean**、属性和方法

SpEL所能做的另外一件基础的事情就是通过ID引用其他的bean。例如，你可以使用SpEL将一个bean装配到另外一个bean的属性中，此时要使用bean ID作为SpEL表达式（在本例中，也就是sgtPeppers）：

```
#{sgtPeppers}
```

现在，假设我们想在一个表达式中引用sgtPeppers的artist属性：

```
#{sgtPeppers.artist}
```

表达式主体的第一部分引用了一个ID为sgtPeppers的bean，分割符之后是对artist属性的引用。

除了引用bean的属性，我们还可以调用bean上的方法。例如，假设有另外一个bean，它的ID为artistSelector，我们可以在SpEL表达式中按照如下的方式来调用bean的selectArtist()方法：

```
#{artistSelector.selectArtist()}
```

对于被调用方法的返回值来说，我们同样可以调用它的方法。例如，如果selectArtist()方法返回的是一个String，那么可以调用toUpperCase()将整个艺术家的名字改为大写字母形式：

```
#{artistSelector.selectArtist().toUpperCase()}
```

如果selectArtist()的返回值不是null的话，这没有什么问题。为了避免出现NullPointerException，我们可以使用类型安全的运算符：

```
#{artistSelector.selectArtist()?.toUpperCase()}
```

与之前只是使用点号（.）来访问toUpperCase()方法不同，现在我们使用了“?.”运算符。这个运算符能够在访问它右边的内容之前，确

保它所对应的元素不是null。所以，如果selectArtist()的返回值是null的话，那么SpEL将不会调用toUpperCase()方法。表达式的返回值会是null。

在表达式中使用类型

如果要在SpEL中访问类作用域的方法和常量的话，要依赖T()这个关键的运算符。例如，为了在SpEL中表达Java的Math类，需要按照如下的方式使用T()运算符：

```
T(java.lang.Math)
```

这里所示的T()运算符的结果会是一个Class对象，代表了java.lang.Math。如果需要的话，我们甚至可以将其装配到一个Class类型的bean属性中。但是T()运算符的真正价值在于它能够访问目标类型的静态方法和常量。

例如，假如你需要将PI值装配到bean属性中。如下的SpEL就能完成该任务：

```
T(java.lang.Math).PI
```

与之类似，我们可以调用T()运算符所得到类型的静态方法。我们已经看到了通过T()调用System.currentTimeMillis()。如下的这个样例会计算得到一个0到1之间的随机数：

```
T(java.lang.Math).random()
```

SpEL运算符

SpEL提供了多个运算符，这些运算符可以用在SpEL表达式的值上。表3.1概述了这些运算符。

表**3.1**　用来操作表达式值的**SpEL**运算符

运算符类型	运　算　符
算术运算	+、-、 * 、/、%、^
比较运算	< 、 > 、 == 、 <= 、 >= 、 lt 、 gt 、 eq 、 le 、 ge

逻辑运算	and 、 or 、 not 、 |
条件运算	?: (ternary) 、 ?: (Elvis)
正则表达式	matches

作为使用上述运算符的一个简单样例，我们看一下下面这个SpEL表达式：

```
#{2 * T(java.lang.Math).PI * circle.radius}
```

这不仅是使用SpEL中乘法运算符（*）的绝佳样例，它也为你展现了如何将简单的表达式组合为更为复杂的表达式。在这里PI的值乘以2，然后再乘以`radius`属性的值，这个属性来源于ID为`circle`的bean。实际上，它计算了`circle` bean中所定义圆的周长。

类似地，你还可以在表达式中使用乘方运算符（^）来计算圆的面积：

```
#{T(java.lang.Math).PI * circle.radius ^ 2}
```

“^”是用于乘方计算的运算符。在本例中，我们使用它来计算圆半径的平方。

当使用String类型的值时，“+”运算符执行的是连接操作，与在Java中是一样的：

```
#{disc.title + ' by ' + disc.artist}
```

SpEL同时还提供了比较运算符，用来在表达式中对值进行对比。注意在表3.1中，比较运算符有两种形式：符号形式和文本形式。在大多数情况下，符号运算符与对应的文本运算符作用是相同的，使用哪一种形式均可以。

例如，要比较两个数字是不是相等，可以使用双等号运算符（==）：

```
#{counter.total == 100}
```

或者，也可以使用文本型的eq运算符：

```
#{counter.total eq 100}
```

两种方式的结果都是一样的。表达式的计算结果是个Boolean值：如果counter.total等于100的话，为true，否则为false。

SpEL还提供了三元运算符（ternary），它与Java中的三元运算符非常类似。例如，如下的表达式会判断如果scoreboard.score>1000的话，计算结果为String类型的“Winner！”，否则的话，结果为Loser：

```
#{scoreboard.score > 1000 ? "Winner!" : "Loser"}
```

三元运算符的一个常见场景就是检查null值，并用一个默认值来替代null。例如，如下的表达式会判断disc.title的值是不是null，如果是null的话，那么表达式的计算结果就会是“Rattle and Hum”：

```
#{disc.title ?: 'Rattle and Hum'}
```

这种表达式通常称为Elvis运算符。这个奇怪名称的来历是，当使用符号来表示表情时，问号看起来很像是猫王（Elvis Presley）的头发。

计算正则表达式

当处理文本时，有时检查文本是否匹配某种模式是非常有用的。SpEL通过matches运算符支持表达式中的模式匹配。matches运算符对String类型的文本（作为左边参数）应用正则表达式（作为右边参数）。matches的运算结果会返回一个Boolean类型的值：如果与正则表达式相匹配，则返回true；否则返回false。

为了进一步解释matches运算符，假设我们想判断一个字符串是否包含有效的邮件地址。在这个场景下，我们可以使用matches运算符，如下所示：

```
#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.com'}
```

探寻正则表达式语法的秘密超出了本书的范围，同时我们也应该意识到这里的正则表达式还不足够健壮来涵盖所有的场景。但对于演示matches运算符的用法，这已经足够了。

计算集合

SpEL中最令人惊奇的一些技巧是与集合和数组相关的。最简单的事情可能就是引用列表中的一个元素了：

```
#{jukebox.songs[4].title}
```

这个表达式会计算songs集合中第五个（基于零开始）元素的title属性，这个集合来源于ID为jukebox bean。

为了让这个表达式更丰富一些，假设我们要从jukebox中随机选择一首歌：

```
#{jukebox.songs[T(java.lang.Math).random() *
                        jukebox.songs.size()].title}
```

“[]”运算符用来从集合或数组中按照索引获取元素，实际上，它还可以从String中获取一个字符。比如：

```
#{'This is a test'[3]}
```

这个表达式引用了String中的第四个（基于零开始）字符，也就是“s”。[2]

SpEL还提供了查询运算符（.?[]），它会用来对集合进行过滤，得到集合的一个子集。作为阐述的样例，假设你希望得到jukebox中artist属性为Aerosmith的所有歌曲。如下的表达式就使用查询运算符得到了Aerosmith的所有歌曲：

```
#{jukebox.songs.?[artist eq 'Aerosmith']}
```

可以看到，选择运算符在它的方括号中接受另一个表达式。当SpEL迭代歌曲列表的时候，会对歌曲集合中的每一个条目计算这个表达式。如果表达式的计算结果为true的话，那么条目会放到新的集合中。否则的话，它就不会放到新集合中。在本例中，内部的表达式会检查歌曲的artist属性是不是等于Aerosmith。

SpEL还提供了另外两个查询运算符：“.^[]”和“.$[]”，它们分别用来在集合中查询第一个匹配项和最后一个匹配项。例如，考虑下面的表达式，它会查找列表中第一个artist属性为Aerosmith的歌曲：

```
#{jukebox.songs.^[artist eq 'Aerosmith']}
```

最后，SpEL还提供了投影运算符（.![]），它会从集合的每个成员中选择特定的属性放到另外一个集合中。作为样例，假设我们不想要歌曲对象的集合，而是所有歌曲名称的集合。如下的表达式会将title属性投影到一个新的String类型的集合中：

```
#{jukebox.songs.![title]}
```

实际上，投影操作可以与其他任意的SpEL运算符一起使用。比如，我们可以使用如下的表达式获得Aerosmith所有歌曲的名称列表：

```
#{jukebox.songs.?[artist eq 'Aerosmith'].![title]}
```

我们所介绍的只是SpEL功能的一个皮毛。在本书中还有更多的机会继续介绍SpEL，尤其是在定义安全规则的时候。

现在对SpEL的介绍要告一段落了，不过在此之前，我们有一个提示。在动态注入值到Spring bean时，SpEL是一种很便利和强大的方式。我们有时会忍不住编写很复杂的表达式。但需要注意的是，不要让你的表达式太智能。你的表达式越智能，对它的测试就越重要。SpEL毕竟只是String类型的值，可能测试起来很困难。鉴于这一点，我建议尽可能让表达式保持简洁，这样测试不会是什么大问题。

3.6 小结

我们在本章介绍了许多背景知识，在第2章所介绍的基本bean装配基础之上，又学习了一些强大的高级装配技巧。

首先，我们学习了Spring profile，它解决了Spring bean要跨各种部署环境的通用问题。在运行时，通过将环境相关的bean与当前激活的profile进行匹配，Spring能够让相同的部署单元跨多种环境运行，而不需要进行重新构建。

Profile bean是在运行时条件化创建bean的一种方式，但是Spring 4提供了一种更为通用的方式，通过这种方式能够声明某些bean的创建与否要依赖于给定条件的输出结果。结合使用@Conditional注解和Spring Condition接口的实现，能够为开发人员提供一种强大和灵活的机制，实现条件化地创建bean。

我们还看了两种解决自动装配歧义性的方法：首选bean以及限定符。尽管将某个bean设置为首选bean是很简单的，但这种方式也有其局限

性，所以我们讨论了如何将一组可选的自动装配bean，借助限定符将其范围缩小到只有一个符合条件的bean。除此之外，我们还看到了如何创建自定义的限定符注解，这些限定符描述了bean的特性。

尽管大多数的Spring bean都是以单例的方式创建的，但有的时候其他的创建策略更为合适。Spring能够让bean以单例、原型、请求作用域或会话作用域的方式来创建。在声明请求作用域或会话作用域的bean的时候，我们还学习了如何创建作用域代理，它分为基于类的代理和基于接口的代理的两种方式。

最后，我们学习了Spring表达式语言，它能够在运行时计算要注入到bean属性中的值。

对于bean装配，我们已经掌握了扎实的基础知识，现在我们要将注意力转向面向切面编程（aspect-oriented programming，AOP）了。依赖注入能够将组件及其协作的其他组件解耦，与之类似，AOP有助于将应用组件与跨多个组件的任务进行解耦。在下一章，我们将会深入学习在Spring中如何创建和使用切面。

---

[1]Java 8允许出现重复的注解，只要这个注解本身在定义的时候带有`@Repeatable`注解就可以。不过，Spring的`@Qualifier`注解并没有在定义时添加`@Repeatable`注解。

[2]不要责怪我，我不太认同这个名字。但是我必须承认，它看起来确实有点像猫王的头发。

第4章　面向切面的Spring

本章内容：

- 面向切面编程的基本原理
- 通过POJO创建切面
- 使用@AspectJ注解
- 为AspectJ切面注入依赖

在编写本章时，得克萨斯州（我所居住的地方）正值盛夏，这几天正在经历创历史记录的高温天气。这里真的非常热，在这种天气下，空调当然是必不可少的。但是空调的缺点是它会耗电，而电需要钱。为了享受凉爽和舒适，我们没有什么办法可以避免这种开销。这是因为每家每户都有一个电表来记录用电量，每个月都会有人来查电表，这样电力公司就知道应该收取多少费用了。

现在想象一下，如果没有电表，也没有人来查看用电量，假设现在由户主来联系电力公司并报告自己的用电量。虽然可能会有一些特别执着的户主会详细记录使用电灯、电视和空调的情况，但大多数人肯定不会这么做。基于信用的电力收费对于消费者可能非常不错，但对于电力公司来说结果可能就不那么美妙了。

监控用电量是一个很重要的功能，但并不是大多数家庭重点关注的问题。所有家庭实际上所关注的可能是修剪草坪、用吸尘器清理地毯、打扫浴室等事项。从家庭的角度来看，监控房屋的用电量是一个被动事件（其实修剪草坪也是一个被动事件——特别是在炎热的天气下）。

软件系统中的一些功能就像我们家里的电表一样。这些功能需要用到应用程序的多个地方，但是我们又不想在每个点都明确调用它们。日志、安全和事务管理的确都很重要，但它们是否为应用对象主动参与的行为呢？如果让应用对象只关注于自己所针对的业务领域问题，而其他方面的问题由其他应用对象来处理，这会不会更好呢？

在软件开发中，散布于应用中多处的功能被称为横切关注点（cross-cutting concern）。通常来讲，这些横切关注点从概念上是与应用的业务逻辑相分离的（但是往往会直接嵌入到应用的业务逻辑之中）。把这些横切关注点与业务逻辑相分离正是面向切面编程（AOP）所要解

决的问题。

在第2章，我们介绍了如何使用依赖注入（DI）管理和配置我们的应用对象。DI有助于应用对象之间的解耦，而AOP可以实现横切关注点与它们所影响的对象之间的解耦。

日志是应用切面的常见范例，但它并不是切面适用的唯一场景。通览本书，我们还会看到切面所适用的多个场景，包括声明式事务、安全和缓存。

本章展示了Spring对切面的支持，包括如何把普通类声明为一个切面和如何使用注解创建切面。除此之外，我们还会看到AspectJ——另一种流行的AOP实现——如何补充Spring AOP框架的功能。但是，我们先不管事务、安全和缓存，先看一下Spring是如何实现切面的，就从AOP的基础知识开始吧。

4.1　什么是面向切面编程

如前所述，切面能帮助我们模块化横切关注点。简而言之，横切关注点可以被描述为影响应用多处的功能。例如，安全就是一个横切关注点，应用中的许多方法都会涉及到安全规则。图4.1直观呈现了横切关注点的概念。

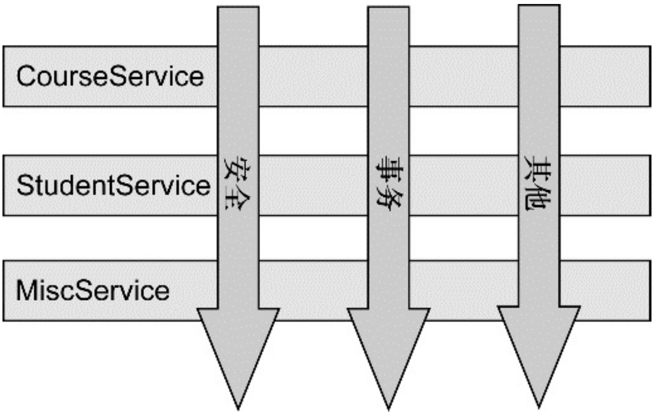

图4.1　切面实现了横切关注点（跨多个应用对象的逻辑）的模块化

图4.1展现了一个被划分为模块的典型应用。每个模块的核心功能都是为特定业务领域提供服务，但是这些模块都需要类似的辅助功能，例如安全和事务管理。

如果要重用通用功能的话，最常见的面向对象技术是继承

（inheritance）或委托（delegation）。但是，如果在整个应用中都使用相同的基类，继承往往会导致一个脆弱的对象体系；而使用委托可能需要对委托对象进行复杂的调用。

切面提供了取代继承和委托的另一种可选方案，而且在很多场景下更清晰简洁。在使用面向切面编程时，我们仍然在一个地方定义通用功能，但是可以通过声明的方式定义这个功能要以何种方式在何处应用，而无需修改受影响的类。横切关注点可以被模块化为特殊的类，这些类被称为切面（aspect）。这样做有两个好处：首先，现在每个关注点都集中于一个地方，而不是分散到多处代码中；其次，服务模块更简洁，因为它们只包含主要关注点（或核心功能）的代码，而次要关注点的代码被转移到切面中了。

4.1.1　定义AOP术语

与大多数技术一样，AOP已经形成了自己的术语。描述切面的常用术语有通知（advice）、切点（pointcut）和连接点（join point）。图4.2展示了这些概念是如何关联在一起的。

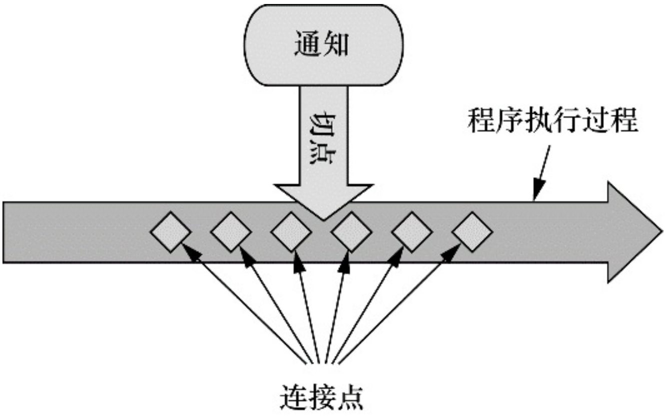

图4.2　在一个或多个连接点上，
可以把切面的功能（通知）织入到程序的执行过程中

遗憾的是，大多数用于描述AOP功能的术语并不直观，尽管如此，它们现在已经是AOP行话的组成部分了，为了理解AOP，我们必须了解这些术语。在我们进入某个领域之前，必须学会在这个领域该如何说话。

通知（**Advice**）

当抄表员出现在我们家门口时，他们要登记用电量并回去向电力公司报告。显然，他们必须有一份需要抄表的住户清单，他们所汇报的信

息也很重要，但记录用电量才是抄表员的主要工作。

类似地，切面也有目标——它必须要完成的工作。在AOP术语中，切面的工作被称为通知。

通知定义了切面是什么以及何时使用。除了描述切面要完成的工作，通知还解决了何时执行这个工作的问题。它应该应用在某个方法被调用之前？之后？之前和之后都调用？还是只在方法抛出异常时调用？

Spring切面可以应用5种类型的通知：

- 前置通知（Before）：在目标方法被调用之前调用通知功能；
- 后置通知（After）：在目标方法完成之后调用通知，此时不会关心方法的输出是什么；
- 返回通知（After-returning）：在目标方法成功执行之后调用通知；
- 异常通知（After-throwing）：在目标方法抛出异常后调用通知；
- 环绕通知（Around）：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的行为。

连接点（**Join point**）

电力公司为多个住户提供服务，甚至可能是整个城市。每家都有一个电表，这些电表上的数字都需要读取，因此每家都是抄表员的潜在目标。抄表员也许能够读取各种类型的设备，但是为了完成他的工作，他的目标应该房屋内所安装的电表。

同样，我们的应用可能也有数以千计的时机应用通知。这些时机被称为连接点。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

切点（**Poincut**）

如果让一位抄表员访问电力公司所服务的所有住户，那肯定是不现实的。实际上，电力公司为每一个抄表员都分别指定某一块区域的住户。类似地，一个切面并不需要通知应用的所有连接点。切点有助于缩小切面所通知的连接点的范围。

如果说通知定义了切面的“什么”和“何时”的话，那么切点就定义了“何处”。切点的定义会匹配通知所要织入的一个或多个连接点。我

们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。有些AOP框架允许我们创建动态的切点，可以根据运行时的决策（比如方法的参数值）来决定是否应用通知。

切面（**Aspect**）

当抄表员开始一天的工作时，他知道自己要做的事情（报告用电量）和从哪些房屋收集信息。因此，他知道要完成工作所需要的一切东西。

切面是通知和切点的结合。通知和切点共同定义了切面的全部内容——它是什么，在何时和何处完成其功能。

引入（**Introduction**）

引入允许我们向现有的类添加新方法或属性。例如，我们可以创建一个`Auditable`通知类，该类记录了对象最后一次修改时的状态。这很简单，只需一个方法，`setLastModified(Date)`，和一个实例变量来保存这个状态。然后，这个新方法和实例变量就可以被引入到现有的类中，从而可以在无需修改这些现有的类的情况下，让它们具有新的行为和状态。

织入（**Weaving**）

织入是把切面应用到目标对象并创建新的代理对象的过程。切面在指定的连接点被织入到目标对象中。在目标对象的生命周期里有多个点可以进行织入：

- 编译期：切面在目标类编译时被织入。这种方式需要特殊的编译器。AspectJ的织入编译器就是以这种方式织入切面的。
- 类加载期：切面在目标类加载到JVM时被织入。这种方式需要特殊的类加载器（`ClassLoader`），它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ 5的加载时织入（load-time weaving，LTW）就支持以这种方式织入切面。
- 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。Spring AOP就是以这种方式织入切面的。

要掌握的新术语可真不少啊。再看一下图4.1，现在我们已经了解了如下的知识，通知包含了需要用于多个应用对象的横切行为；连接点

是程序执行过程中能够应用通知的所有点；切点定义了通知被应用的具体位置（在哪些连接点）。其中关键的概念是切点定义了哪些连接点会得到通知。

我们已经了解了一些基础的AOP术语，现在让我们再看看这些AOP的核心概念是如何在Spring中实现的。

4.1.2 Spring对AOP的支持

并不是所有的AOP框架都是相同的，它们在连接点模型上可能有强弱之分。有些允许在字段修饰符级别应用通知，而另一些只支持与方法调用相关的连接点。它们织入切面的方式和时机也有所不同。但是无论如何，创建切点来定义切面所织入的连接点是AOP框架的基本功能。

因为这是一本介绍Spring的图书，所以我们会关注Spring AOP。虽然如此，Spring和AspectJ项目之间有大量的协作，而且Spring对AOP的支持在很多方面借鉴了AspectJ项目。

Spring提供了4种类型的AOP支持：

- 基于代理的经典Spring AOP；
- 纯POJO切面；
- `@AspectJ`注解驱动的切面；
- 注入式AspectJ切面（适用于Spring各版本）。

前三种都是Spring AOP实现的变体，Spring AOP构建在动态代理基础之上，因此，Spring对AOP的支持局限于方法拦截。

术语“经典”通常意味着是很好的东西。老爷车、经典高尔夫球赛、可口可乐精品都是好东西。但是Spring的经典AOP编程模型并不怎么样。当然，曾经它的确非常棒。但是现在Spring提供了更简洁和干净的面向切面编程方式。引入了简单的声明式AOP和基于注解的AOP之后，Spring经典的AOP看起来就显得非常笨重和过于复杂，直接使用ProxyFactory Bean会让人感觉厌烦。所以在本书中我不会再介绍经典的Spring AOP。

借助Spring的`aop`命名空间，我们可以将纯POJO转换为切面。实际上，这些POJO只是提供了满足切点条件时所要调用的方法。遗憾的是，这种技术需要XML配置，但这的确是声明式地将对象转换为切面的简便方式。

Spring借鉴了AspectJ的切面，以提供注解驱动的AOP。本质上，它依然是Spring基于代理的AOP，但是编程模型几乎与编写成熟的AspectJ注解切面完全一致。这种AOP风格的好处在于能够不使用XML来完成功能。

如果你的AOP需求超过了简单的方法调用（如构造器或属性拦截），那么你需要考虑使用AspectJ来实现切面。在这种情况下，上文所示的第四种类型能够帮助你将值注入到AspectJ驱动的切面中。

我们在将在本章展示更多的Spring AOP技术，但是在开始之前，我们必须要了解Spring AOP框架的一些关键知识。

Spring通知是**Java**编写的

Spring所创建的通知都是用标准的Java类编写的。这样的话，我们就可以使用与普通Java开发一样的集成开发环境（IDE）来开发切面。而且，定义通知所应用的切点通常会使用注解或在Spring配置文件里采用XML来编写，这两种语法对于Java开发者来说都是相当熟悉的。

AspectJ与之相反。虽然AspectJ现在支持基于注解的切面，但AspectJ最初是以Java语言扩展的方式实现的。这种方式有优点也有缺点。通过特有的AOP语言，我们可以获得更强大和细粒度的控制，以及更丰富的AOP工具集，但是我们需要额外学习新的工具和语法。

Spring在运行时通知对象

通过在代理类中包裹切面，Spring在运行期把切面织入到Spring管理的bean中。如图4.3所示，代理类封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的目标bean。当代理拦截到方法调用时，在调用目标bean方法之前，会执行切面逻辑。

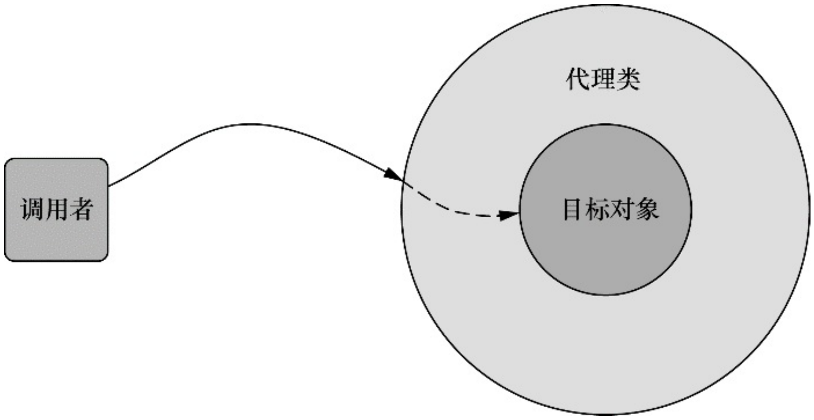

图4.3　Spring的切面由包裹了目标对象的代理类实现。
代理类处理方法的调用，执行额外的切面逻辑，并调用目标方法

直到应用需要被代理的bean时，Spring才创建代理对象。如果使用的是ApplicationContext的话，在ApplicationContext从BeanFactory中加载所有bean的时候，Spring才会创建被代理的对象。因为Spring运行时才创建代理对象，所以我们不需要特殊的编译器来织入Spring AOP的切面。

Spring只支持方法级别的连接点

正如前面所探讨过的，通过使用各种AOP方案可以支持多种连接点模型。因为Spring基于动态代理，所以Spring只支持方法连接点。这与一些其他的AOP框架是不同的，例如AspectJ和JBoss，除了方法切点，它们还提供了字段和构造器接入点。Spring缺少对字段连接点的支持，无法让我们创建细粒度的通知，例如拦截对象字段的修改。而且它不支持构造器连接点，我们就无法在bean创建时应用通知。

但是方法拦截可以满足绝大部分的需求。如果需要方法拦截之外的连接点拦截功能，那么我们可以利用Aspect来补充Spring AOP的功能。

对于什么是AOP以及Spring如何支持AOP的，我们现在已经有了一个大致的了解。现在是时候学习如何在Spring中创建切面了，让我们先从Spring的声明式AOP模型开始。

4.2　通过切点来选择连接点

正如之前所提过的，切点用于准确定位应该在什么地方应用切面的通知。通知和切点是切面的最基本元素。因此，了解如何编写切点非常重要。

在Spring AOP中，要使用AspectJ的切点表达式语言来定义切点。如果你已经很熟悉AspectJ，那么在Spring中定义切点就感觉非常自然。但是如果你一点都不了解AspectJ的话，本小节我们将快速介绍一下如何编写AspectJ风格的切点。如果你想进一步了解AspectJ和AspectJ切点表达式语言，我强烈推荐Ramniva Laddad编写的《AspectJ in Action》第二版（Manning，2009，www.manning.com/laddad2/）。

关于Spring AOP的AspectJ切点，最重要的一点就是Spring仅支持AspectJ切点指示器（pointcut designator）的一个子集。让我们回顾

下，Spring是基于代理的，而某些切点表达式是与基于代理的AOP无关的。表4.1列出了Spring AOP所支持的AspectJ切点指示器。

表**4.1　Spring**借助**AspectJ**的切点表达式语言来定义**Spring**切面

AspectJ指示器	描　　述
`arg()`	限制连接点匹配参数为指定类型的执行方法
`@args()`	限制连接点匹配参数由指定注解标注的执行方法
`execution()`	用于匹配是连接点的执行方法
`this()`	限制连接点匹配AOP代理的bean引用为指定类型的类
`target`	限制连接点匹配目标对象为指定类型的类
`@target()`	限制连接点匹配特定的执行对象，这些对象对应的类要具有指定类型的注解
`within()`	限制连接点匹配指定的类型
`@within()`	限制连接点匹配指定注解所标注的类型（当使用Spring AOP时，方法定义在由指定的注解所标注的类里）
`@annotation`	限定匹配带有指定注解的连接点

在Spring中尝试使用AspectJ其他指示器时，将会抛出`IllegalArgument-Exception`异常。

当我们查看如上所展示的这些Spring支持的指示器时，注意只有`execution`指示器是实际执行匹配的，而其他的指示器都是用来限制匹配的。这说明`execution`指示器是我们在编写切点定义时最主要使用的指示器。在此基础上，我们使用其他指示器来限制所匹配

的切点。

4.2.1 编写切点

为了阐述Spring中的切面，我们需要有个主题来定义切面的切点。为此，我们定义一个Performance接口：

```
package concert;

public interface Performance {
  public void perform();
}
```

Performance可以代表任何类型的现场表演，如舞台剧、电影或音乐会。假设我们想编写Performance的perform()方法触发的通知。图4.4展现了一个切点表达式，这个表达式能够设置当perform()方法执行时触发通知的调用。

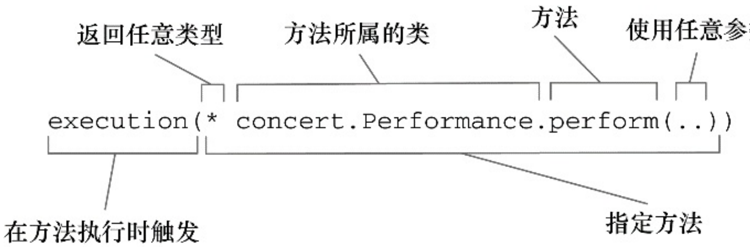

图4.4　使用AspectJ切点表达式来选择Performance的perform()方法

我们使用execution()指示器选择Performance的perform()方法。方法表达式以“*”号开始，表明了我们不关心方法返回值的类型。然后，我们指定了全限定类名和方法名。对于方法参数列表，我们使用两个点号（..）表明切点要选择任意的perform()方法，无论该方法的入参是什么。

现在假设我们需要配置的切点仅匹配concert包。在此场景下，可以使用within()指示器来限制匹配，如图4.5所示。

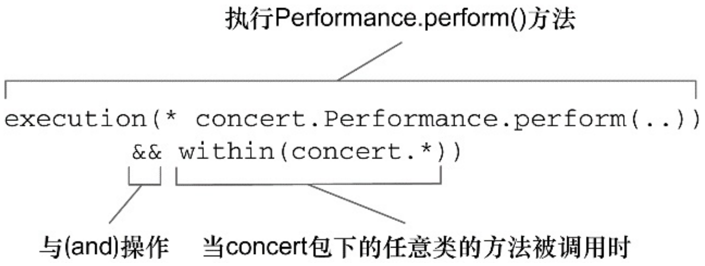

图4.5　使用within()指示器限制切点范围

请注意我们使用了“&&”操作符把execution()和within()指示器连接在一起形成与（and）关系（切点必须匹配所有的指示器）。类似地，我们可以使用“||”操作符来标识或（or）关系，而使用“!”操作符来标识非（not）操作。

因为“&”在XML中有特殊含义，所以在Spring的XML配置里面描述切点时，我们可以使用and来代替“&&”。同样，or和not可以分别用来代替“||”和“!”。

4.2.2 在切点中选择bean

除了表4.1所列的指示器外，Spring还引入了一个新的bean()指示器，它允许我们在切点表达式中使用bean的ID来标识bean。bean()使用bean ID或bean名称作为参数来限制切点只匹配特定的bean。

例如，考虑如下的切点：

```
execution(* concert.Performance.perform())
        and bean('woodstock')
```

在这里，我们希望在执行Performance的perform()方法时应用通知，但限定bean的ID为woodstock。

在某些场景下，限定切点为指定的bean或许很有意义，但我们还可以使用非操作为除了特定ID以外的其他bean应用通知：

```
execution(* concert.Performance.perform())
        and !bean('woodstock')
```

在此场景下，切面的通知会被编织到所有ID不为woodstock的bean中。

现在，我们已经讲解了编写切点的基础知识，让我们再了解一下如何编写通知和使用这些切点声明切面。

4.3 使用注解创建切面

使用注解来创建切面是AspectJ 5所引入的关键特性。AspectJ 5之前，编写AspectJ切面需要学习一种Java语言的扩展，但是AspectJ面向注解的模型可以非常简便地通过少量注解把任意类转变为切面。

我们已经定义了Performance接口，它是切面中切点的目标对象。现在，让我们使用AspecJ注解来定义切面。

4.3.1 定义切面

如果一场演出没有观众的话，那不能称之为演出。对不对？从演出的角度来看，观众是非常重要的，但是对演出本身的功能来讲，它并不是核心，这是一个单独的关注点。因此，将观众定义为一个切面，并将其应用到演出上就是较为明智的做法。

程序清单4.1展现了Audience类，它定义了我们所需的一个切面。

程序清单**4.1 Audience**类：观看演出的切面

```
package concert;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class Audience {
                                                                        表演之前
  @Before("execution(** concert.Performance.perform(..))")
  public void silenceCellPhones() {

    System.out.println("Silencing cell phones");
  }
                                                                        表演之前
  @Before("execution(** concert.Performance.perform(..))")
  public void takeSeats() {
    System.out.println("Taking seats");
  }

  @AfterReturning("execution(** concert.Performance.perform(..))")
   public void applause() {
    System.out.println("CLAP CLAP CLAP!!!");                            表演之后
  }

  @AfterThrowing("execution(** concert.Performance.perform(..))")
   public void demandRefund() {
    System.out.println("Demanding a refund");                           表演失败之后
  }

}
```

Audience类使用@AspectJ注解进行了标注。该注解表明Audience不仅仅是一个POJO，还是一个切面。Audience类中的方法都使用注解来定义切面的具体行为。

Audience有四个方法，定义了一个观众在观看演出时可能会做的事情。在演出之前，观众要就坐（takeSeats()）并将手机调至静音状态（silenceCellPhones()）。如果演出很精彩的话，观众应该会鼓掌喝彩（applause()）。不过，如果演出没有达到观众预期的话，观众会要求退款（demandRefund()）。

可以看到，这些方法都使用了通知注解来表明它们应该在什么时候调用。AspectJ提供了五个注解来定义通知，如表4.2所示。

表**4.2　Spring**使用**AspectJ**注解来声明通知方法

注　　解	通　　知
@After	通知方法会在目标方法返回或抛出异常后调用
@AfterReturning	通知方法会在目标方法返回后调用
@AfterThrowing	通知方法会在目标方法抛出异常后调用
@Around	通知方法会将目标方法封装起来
@Before	通知方法会在目标方法调用之前执行

Audience使用到了前面五个注解中的三个。takeSeats()和silence CellPhones()方法都用到了@Before注解，表明它们应该在演出开始之前调用。applause()方法使用了@AfterReturning注解，它会在演出成功返回后调用。demandRefund()方法上添加了@AfterThrowing注解，这表明它会在抛出异常以后执行。

你可能已经注意到了，所有的这些注解都给定了一个切点表达式作为它的值，同时，这四个方法的切点表达式都是相同的。其实，它们可以设置成不同的切点表达式，但是在这里，这个切点表达式就能满足所有通知方法的需求。让我们近距离看一下这个设置给通知注解的切点表达式，我们发现它会在Performance的perform()方法执行时触发。

相同的切点表达式我们重复了四遍，这可真不是什么光彩的事情。这样的重复让人感觉有些不对劲。如果我们只定义这个切点一次，然后每次需要的时候引用它，那么这会是一个很好的方案。

幸好，我们完全可以这样做：@Pointcut注解能够在一个@AspectJ切面内定义可重用的切点。接下来的程序清单4.2展现了

新的Audience，现在它使用了@Pointcut。

程序清单**4.2**　通过**@Pointcut**注解声明频繁使用的切点表达式

```
package concert;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {

  @Pointcut("execution(** concert.Performance.perform(..))")    定义命名的切点
  public void performance() {}

  @Before("performance()")
  public void silenceCellPhones() {
    System.out.println("Silencing cell phones");              表演之前
  }

  @Before("performance()")
  public void takeSeats() {
    System.out.println("Taking seats");
  }

  @AfterReturning("performance()")                            表演之后
  public void applause() {
    System.out.println("CLAP CLAP CLAP!!!");
  }

  @AfterThrowing("performance()")                             表演失败之后
  public void demandRefund() {
    System.out.println("Demanding a refund");
  }

}
```

在Audience中，performance()方法使用了@Pointcut注解。为@Pointcut注解设置的值是一个切点表达式，就像之前在通知注解上所设置的那样。通过在performance()方法上添加@Pointcut注解，我们实际上扩展了切点表达式语言，这样就可以在任何的切点表达式中使用performance()了，如果不这样做的话，你需要在这些地方使用那个更长的切点表达式。我们现在把所有通知注解中的长表达式都替换成了performance()。

performance()方法的实际内容并不重要，在这里它实际上应该是空的。其实该方法本身只是一个标识，供@Pointcut注解依附。

需要注意的是，除了注解和没有实际操作的performance()方法，Audience类依然是一个POJO。我们能够像使用其他的Java类那样调用它的方法，它的方法也能够独立地进行单元测试，这与其他的Java类并没有什么区别。Audience只是一个Java类，只不过它通过注解表明会作为切面使用而已。

像其他的Java类一样，它可以装配为Spring中的bean：

```
@Bean
public Audience audience() {
    return new Audience();
}
```

如果你就此止步的话，Audience只会是Spring容器中的一个bean。即便使用了AspectJ注解，但它并不会被视为切面，这些注解不会解析，也不会创建将其转换为切面的代理。

如果你使用JavaConfig的话，可以在配置类的类级别上通过使用EnableAspectJ-AutoProxy注解启用自动代理功能。程序清单4.3展现了如何在JavaConfig中启用自动代理。

程序清单**4.3**　在**JavaConfig**中启用**AspectJ**注解的自动代理

```
package concert;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy                    <-- 启用 AspectJ 自动代理
@ComponentScan
public class ConcertConfig {

  @Bean
  public Audience audience() {             <-- 声明 Audience bean
    return new Audience();
  }

}
```

假如你在Spring中要使用XML来装配bean的话，那么需要使用Spring aop命名空间中的<aop:aspectj-autoproxy>元素。下面的XML配置展现了如何完成该功能。

程序清单**4.4**　在**XML**中，通过**Spring**的**aop**命名空间启用**AspectJ**自动代理

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:aop="http://www.springframework.org/schema/aop"        <—— 声明Spring的aop命名空间
  xsi:schemaLocation="http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="concert" />

  <aop:aspectj-autoproxy />        <—— 启用AspectJ自动代理

  <bean class="concert.Audience" />        <—— 声明Audience bean

</beans>
```

不管你是使用JavaConfig还是XML，AspectJ自动代理都会为使用`@Aspect`注解的bean创建一个代理，这个代理会围绕着所有该切面的切点所匹配的bean。在这种情况下，将会为`Concert`bean创建一个代理，`Audience`类中的通知方法将会在`perform()`调用前后执行。

我们需要记住的是，Spring的AspectJ自动代理仅仅使用`@AspectJ`作为创建切面的指导，切面依然是基于代理的。在本质上，它依然是Spring基于代理的切面。这一点非常重要，因为这意味着尽管使用的是`@AspectJ`注解，但我们仍然限于代理方法的调用。如果想利用AspectJ的所有能力，我们必须在运行时使用AspectJ并且不依赖Spring来创建基于代理的切面。

到现在为止，我们的切面在定义时，使用了不同的通知方法来实现前置通知和后置通知。但是表4.2还提到了另外的一种通知：环绕通知（around advice）。环绕通知与其他类型的通知有所不同，因此值得花点时间来介绍如何进行编写。

4.3.2　创建环绕通知

环绕通知是最为强大的通知类型。它能够让你所编写的逻辑将被通知的目标方法完全包装起来。实际上就像在一个通知方法中同时编写前置通知和后置通知。

为了阐述环绕通知，我们重写`Audience`切面。这次，我们使用一个环绕通知来代替之前多个不同的前置通知和后置通知。

程序清单**4.5**　使用环绕通知重新实现**Audience**切面

```java
package concert;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {

  @Pointcut("execution(** concert.Performance.perform(..))")
  public void performance() {}

  @Around("performance()")
  public void watchPerformance(ProceedingJoinPoint jp) {
    try {
      System.out.println("Silencing cell phones");
      System.out.println("Taking seats");
      jp.proceed();
      System.out.println("CLAP CLAP CLAP!!!");
    } catch (Throwable e) {
      System.out.println("Demanding a refund");
    }

  }

}
```

定义命名的切点

环绕通知方法

在这里，@Around注解表明watchPerformance()方法会作为performance()切点的环绕通知。在这个通知中，观众在演出之前会将手机调至静音并就坐，演出结束后会鼓掌喝彩。像前面一样，如果演出失败的话，观众会要求退款。

可以看到，这个通知所达到的效果与之前的前置通知和后置通知是一样的。但是，现在它们位于同一个方法中，不像之前那样分散在四个不同的通知方法里面。

关于这个新的通知方法，你首先注意到的可能是它接受ProceedingJoinPoint作为参数。这个对象是必须要有的，因为你要在通知中通过它来调用被通知的方法。通知方法中可以做任何的事情，当要将控制权交给被通知的方法时，它需要调用ProceedingJoinPoint的proceed()方法。

需要注意的是，别忘记调用proceed()方法。如果不调这个方法的话，那么你的通知实际上会阻塞对被通知方法的调用。有可能这就是你想要的效果，但更多的情况是你希望在某个点上执行被通知的方法。

有意思的是，你可以不调用proceed()方法，从而阻塞对被通知方法的访问，与之类似，你也可以在通知中对它进行多次调用。要这样做的一个场景就是实现重试逻辑，也就是在被通知方法失败后，进行重复尝试。

4.3.3 处理通知中的参数

到目前为止，我们的切面都很简单，没有任何参数。唯一的例外是我们为环绕通知所编写的watchPerformance()示例方法中使用了ProceedingJoinPoint作为参数。除了环绕通知，我们编写的其他通知不需要关注传递给被通知方法的任意参数。这很正常，因为我们所通知的perform()方法本身没有任何参数。

但是，如果切面所通知的方法确实有参数该怎么办呢？切面能访问和使用传递给被通知方法的参数吗？

为了阐述这个问题，让我们重新看一下2.4.4小节中的BlankDisc样例。play()方法会循环所有的磁道并调用playTrack()方法。但是，我们也可以通过playTrack()方法直接播放某一个磁道中的歌曲。

假设你想记录每个磁道被播放的次数。一种方法就是修改playTrack()方法，直接在每次调用的时候记录这个数量。但是，记录磁道的播放次数与播放本身是不同的关注点，因此不应该属于playTrack()方法。看起来，这应该是切面要完成的任务。

为了记录每个磁道所播放的次数，我们创建了TrackCounter类，它是通知playTrack()方法的一个切面。下面的程序清单展示了这个切面。

程序清单**4.6**　使用参数化的通知来记录磁道播放的次数

```
package soundsystem;
import java.util.HashMap;
import java.util.Map;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class TrackCounter {

  private Map<Integer, Integer> trackCounts =
      new HashMap<Integer, Integer>();

  @Pointcut(
    "execution(* soundsystem.CompactDisc.playTrack(int)) " +
    "&& args(trackNumber)")
  public void trackPlayed(int trackNumber) {}

  @Before("trackPlayed(trackNumber)")
  public void countTrack(int trackNumber) {
    int currentCount = getPlayCount(trackNumber);
    trackCounts.put(trackNumber, currentCount + 1);
  }

  public int getPlayCount(int trackNumber) {
    return trackCounts.containsKey(trackNumber)
        ? trackCounts.get(trackNumber) : 0;
  }
}
```

通知 play-Track()方法

在播放前，为该磁道计数

像之前所创建的切面一样，这个切面使用@Pointcut注解定义命名的切点，并使用@Before将一个方法声明为前置通知。但是，这里的不同点在于切点还声明了要提供给通知方法的参数。图4.6将切点表达式进行了分解，以展现参数是在什么地方指定的。

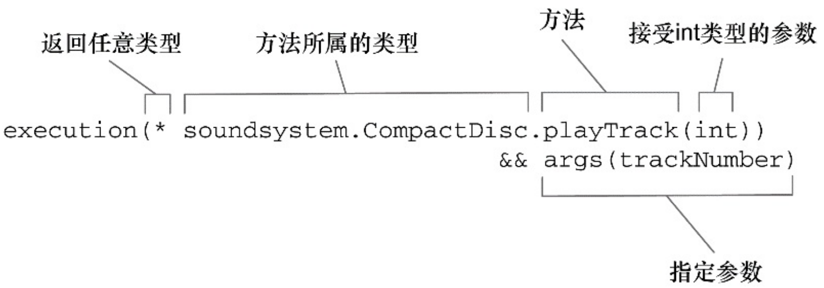

图4.6　在切点表达式中声明参数，这个参数传入到通知方法中

在图4.6中需要关注的是切点表达式中的args(trackNumber)限定符。它表明传递给playTrack()方法的int类型参数也会传递到通知中去。参数的名称trackNumber也与切点方法签名中的参数相匹配。

这个参数会传递到通知方法中，这个通知方法是通过@Before注解和命名切点trackPlayed(trackNumber)定义的。切点定义中的参数与切点方法中的参数名称是一样的，这样就完成了从命名切点到通知方法的参数转移。

现在，我们可以在Spring配置中将BlankDisc和TrackCounter定义为bean，并启用AspectJ自动代理，如程序清单4.7所示。

程序清单**4.7**　配置**TrackCount**记录每个磁道播放的次数

```
package soundsystem;
import java.util.ArrayList;
import java.util.List;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy                    <-- 启用 AspectJ 自动代理
public class TrackCounterConfig {

  @Bean
  public CompactDisc sgtPeppers() {        <-- CompactDisc bean
    BlankDisc cd = new BlankDisc();
    cd.setTitle("Sgt. Pepper's Lonely Hearts Club Band");
    cd.setArtist("The Beatles");
    List<String> tracks = new ArrayList<String>();
    tracks.add("Sgt. Pepper's Lonely Hearts Club Band");
    tracks.add("With a Little Help from My Friends");
    tracks.add("Lucy in the Sky with Diamonds");
    tracks.add("Getting Better");
    tracks.add("Fixing a Hole");

    // ...other tracks omitted for brevity...
    cd.setTracks(tracks);
    return cd;
  }

  @Bean
  public TrackCounter trackCounter() {     <-- TrackCounter bean
    return new TrackCounter();
  }

}
```

最后，为了证明它能正常工作，你可以编写如下的简单测试。它会播放几个磁道并通过`TrackCounter`断言播放的数量。

程序清单**4.8**　测试**TrackCounter**切面

```
package soundsystem;
import static org.junit.Assert.*;
import org.junit.Assert;
import org.junit.Rule;
import org.junit.Test;
import org.junit.contrib.java.lang.system.StandardOutputStreamLog;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=TrackCounterConfig.class)
public class TrackCounterTest {

  @Rule
  public final StandardOutputStreamLog log =
                                     new StandardOutputStreamLog();

  @Autowired
  private CompactDisc cd;

  @Autowired
  private TrackCounter counter;

  @Test
  public void testTrackCounter() {
    cd.playTrack(1);              <————— 播放一些磁道
    cd.playTrack(2);
    cd.playTrack(3);
    cd.playTrack(3);
    cd.playTrack(3);
    cd.playTrack(3);
    cd.playTrack(7);
    cd.playTrack(7);

    assertEquals(1,  counter.getPlayCount(1));     <— 断言期望的数量
    assertEquals(1,  counter.getPlayCount(2));
    assertEquals(4,  counter.getPlayCount(3));
    assertEquals(0,  counter.getPlayCount(4));

    assertEquals(0,  counter.getPlayCount(5));
    assertEquals(0,  counter.getPlayCount(6));
    assertEquals(2,  counter.getPlayCount(7));
  }

}
```

到目前为止，在我们所使用的切面中，所包装的都是被通知对象的已有方法。但是，方法包装仅仅是切面所能实现的功能之一。让我们看一下如何通过编写切面，为被通知的对象引入全新的功能。

4.3.4　通过注解引入新功能

一些编程语言，例如Ruby和Groovy，有开放类的理念。它们可以不用直接修改对象或类的定义就能够为对象或类增加新的方法。不过，Java并不是动态语言。一旦类编译完成了，我们就很难再为该类添加新的功能了。

但是如果仔细想想，我们在本章中不是一直在使用切面这样做吗？当然，我们还没有为对象增加任何新的方法，但是已经为对象拥有的方

法添加了新功能。如果切面能够为现有的方法增加额外的功能，为什么不能为一个对象增加新的方法呢？实际上，利用被称为引入的AOP概念，切面可以为Spring bean添加新方法。

回顾一下，在Spring中，切面只是实现了它们所包装bean相同接口的代理。如果除了实现这些接口，代理也能暴露新接口的话，会怎么样呢？那样的话，切面所通知的bean看起来像是实现了新的接口，即便底层实现类并没有实现这些接口也无所谓。图4.7展示了它们是如何工作的。

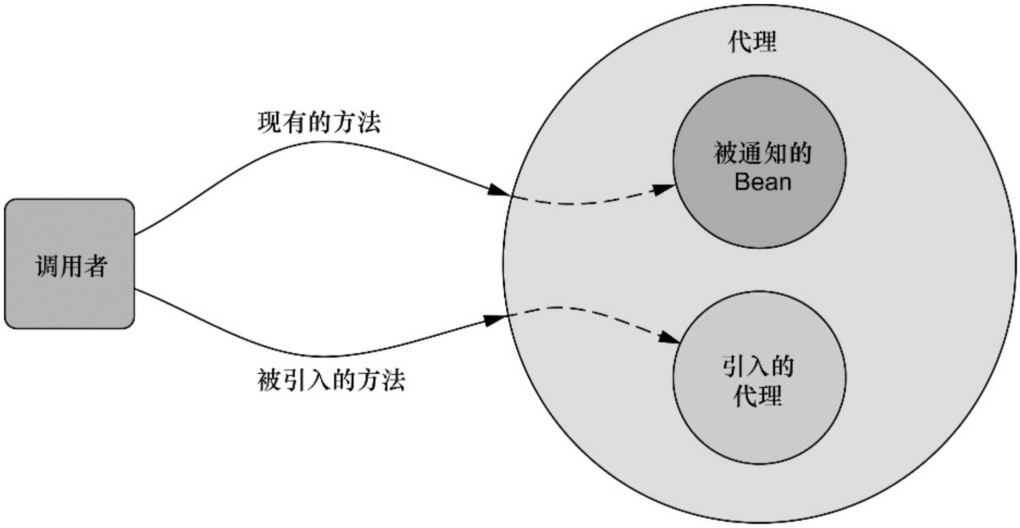

图4.7　使用Spring AOP，我们可以为bean引入新的方法。代理拦截调用并委托给实现该方法的其他对象

我们需要注意的是，当引入接口的方法被调用时，代理会把此调用委托给实现了新接口的某个其他对象。实际上，一个bean的实现被拆分到了多个类中。

为了验证该主意能行得通，我们为示例中的所有的Performance实现引入下面的Encoreable接口：

```
package concert;

public interface Encoreable {
    void performEncore();
}
```

暂且先不管Encoreable是不是一个真正存在的单词[1]，我们需要有一种方式将这个接口应用到Performance实现中。我们现在假设你能够访问Performance的所有实现，并对其进行修改，让它们都实现Encoreable接口。但是，从设计的角度来看，这并不是最好的做法，并不是所有的Performance都是具有Encoreable特性的。另

外一方面，有可能无法修改所有的Performance实现，当使用第三方实现并且没有源码的时候更是如此。

值得庆幸的是，借助于AOP的引入功能，我们可以不必在设计上妥协或者侵入性地改变现有的实现。为了实现该功能，我们要创建一个新的切面：

```
package concert;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class EncoreableIntroducer {

  @DeclareParents(value="concert.Performance+",
                  defaultImpl=DefaultEncoreable.class)
  public static Encoreable encoreable;

}
```

可以看到，EncoreableIntroducer是一个切面。但是，它与我们之前所创建的切面不同，它并没有提供前置、后置或环绕通知，而是通过@DeclareParents注解，将Encoreable接口引入到Performance bean中。

@DeclareParents注解由三部分组成：

- value属性指定了哪种类型的bean要引入该接口。在本例中，也就是所有实现Performance的类型。（标记符后面的加号表示是Performance的所有子类型，而不是Performance本身。）
- defaultImpl属性指定了为引入功能提供实现的类。在这里，我们指定的是DefaultEncoreable提供实现。
- @DeclareParents注解所标注的静态属性指明了要引入了接口。在这里，我们所引入的是Encoreable接口。

和其他的切面一样，我们需要在Spring应用中将EncoreableIntroducer声明为一个bean：

```
<bean class="concert.EncoreableIntroducer" />
```

Spring的自动代理机制将会获取到它的声明，当Spring发现一个bean使用了@Aspect注解时，Spring就会创建一个代理，然后将调用委托给被代理的bean或被引入的实现，这取决于调用的方法属于被代理的bean还是属于被引入的接口。

在Spring中，注解和自动代理提供了一种很便利的方式来创建切面。它非常简单，并且只涉及到最少的Spring配置。但是，面向注解的切面声明有一个明显的劣势：你必须能够为通知类添加注解。为了做到这一点，必须要有源码。

如果你没有源码的话，或者不想将AspectJ注解放到你的代码之中，Spring为切面提供了另外一种可选方案。让我们看一下如何在Spring XML配置文件中声明切面。

4.4 在XML中声明切面

在本书前面的内容中，我曾经建立过这样一种原则，那就是基于注解的配置要优于基于Java的配置，基于Java的配置要优于基于XML的配置。但是，如果你需要声明切面，但是又不能为通知类添加注解的时候，那么就必须转向XML配置了。

在Spring的aop命名空间中，提供了多个元素用来在XML中声明切面，如表4.3所示。

表**4.3** **Spring**的**AOP**配置元素能够以非侵入性的方式声明切面

AOP配置元素	用　　途
`<aop:advisor>`	定义AOP通知器
`<aop:after>`	定义AOP后置通知（不管被通知的方法是否执行成功）
`<aop:after-returning>`	定义AOP返回通知
`<aop:after-throwing>`	定义AOP异常通知
`<aop:around>`	定义AOP环绕通知
`<aop:aspect>`	定义一个切面

<aop:aspectj-autoproxy>	启用@AspectJ注解驱动的切面
<aop:before>	定义一个AOP前置通知
<aop:config>	顶层的AOP配置元素。大多数的<aop:*>元素必须包含在<aop:config>元素内
<aop:declare-parents>	以透明的方式为被通知的对象引入额外的接口
<aop:pointcut>	定义一个切点

我们已经看过了<aop:aspectj-autoproxy>元素，它能够自动代理AspectJ注解的通知类。aop命名空间的其他元素能够让我们直接在Spring配置中声明切面，而不需要使用注解。

例如，我们重新看一下Audience类，这一次我们将它所有的AspectJ注解全部移除掉：

```
package concert;

public class Audience {

  public void silenceCellPhones() {
    System.out.println("Silencing cell phones");
  }

  public void takeSeats() {
    System.out.println("Taking seats");
  }

  public void applause() {
    System.out.println("CLAP CLAP CLAP!!!");
  }

  public void demandRefund() {
    System.out.println("Demanding a refund");
  }

}
```

正如你所看到的，Audience类并没有任何特别之处，它就是有几个方法的简单Java类。我们可以像其他类一样把它注册为Spring应用上下文中的bean。

尽管看起来并没有什么差别，但Audience已经具备了成为AOP通知的所有条件。我们再稍微帮助它一把，它就能够成为预期的通知了。

4.4.1　声明前置和后置通知

你可以再把那些AspectJ注解加回来，但这并不是本节的目的。相反，我们会使用Spring aop命名空间中的一些元素，将没有注解的Audience类转换为切面。下面的程序清单4.9展示了所需要的XML。

程序清单**4.9**　通过**XML**将无注解的**Audience**声明为切面

```
<aop:config>
  <aop:aspect ref="audience">              <-- 引用 audience Bean

    <aop:before
      pointcut="execution(** concert.Performance.perform(..))"
      method="silenceCellPhones"/>                                  表演之前

    <aop:before
      pointcut="execution(** concert.Performance.perform(..))"
      method="takeSeats"/>

    <aop:after-returning                   <-- 表演之后
      pointcut="execution(** concert.Performance.perform(..))"
      method="applause"/>

    <aop:after-throwing                    <-- 表演失败之后
      pointcut="execution(** concert.Performance.perform(..))"
      method="demandRefund"/>

  </aop:aspect>
</aop:config>
```

关于Spring AOP配置元素，第一个需要注意的事项是大多数的AOP配置元素必须在<aop:config>元素的上下文内使用。这条规则有几种例外场景，但是把bean声明为一个切面时，我们总是从<aop:config>元素开始配置的。

在<aop:config>元素内，我们可以声明一个或多个通知器、切面或者切点。在程序清单4.9中，我们使用<aop:aspect>元素声明了一个简单的切面。ref元素引用了一个POJO bean，该bean实现了切面的功能——在这里就是audience。ref元素所引用的bean提供了在切面中通知所调用的方法。

该切面应用了四个不同的通知。两个<aop:before>元素定义了匹配切点的方法执行之前调用前置通知方法—也就是Audience bean的takeSeats()和turnOffCellPhones()方法（由method属性所声明）。<aop:after-returning>元素定义了一个返回（after-returning）通知，在切点所匹配的方法调用之后再调用applaud()方法。同样，<aop:after-throwing>元素定义了异常（after-throwing）通知，如果所匹配的方法执行时抛出任何的异常，都将会调用demandRefund()方法。图4.8展示了通知逻

辑如何织入到业务逻辑中。

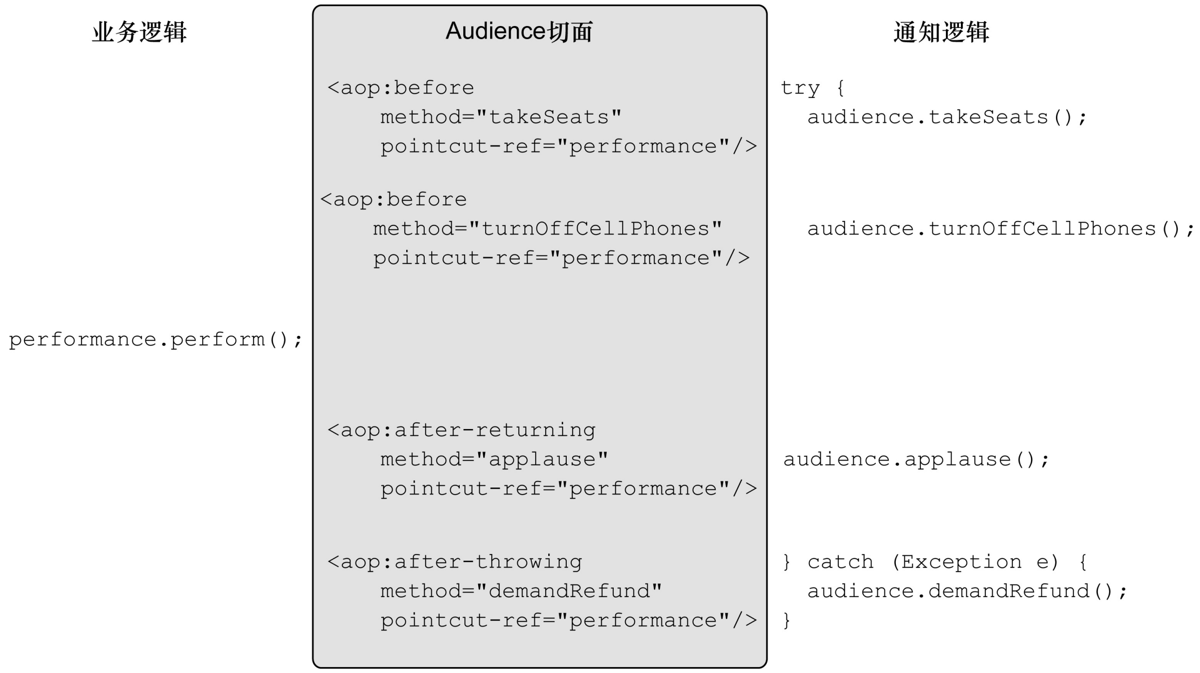

图4.8　Audience切面包含四种通知，它们把通知逻辑织入进匹配切面切点的方法中

在所有的通知元素中，`pointcut`属性定义了通知所应用的切点，它的值是使用AspectJ切点表达式语法所定义的切点。

你或许注意到所有通知元素中的`pointcut`属性的值都是一样的，这是因为所有的通知都要应用到相同的切点上。

在基于AspectJ注解的通知中，当发现这种类型的重复时，我们使用`@Pointcut`注解消除了这些重复的内容。而在基于XML的切面声明中，我们需要使用`<aop:pointcut>`元素。如下的XML展现了如何将通用的切点表达式抽取到一个切点声明中，这样这个声明就能在所有的通知元素中使用了。

程序清单**4.10**　使用**<aop:pointcut>**定义命名切点

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut                                  定义切点
        id="performance"
        expression="execution(** concert.Performance.perform(..))" />
    <aop:before
      pointcut-ref="performance"
      method="silenceCellPhones"/>

    <aop:before                                    引用切点
      pointcut-ref="performance"
      method="takeSeats"/>

    <aop:after-returning
      pointcut-ref="performance"
      method="applause"/>

    <aop:after-throwing                            引用切点
      pointcut-ref="performance"
      method="demandRefund"/>
  </aop:aspect>
</aop:config>
```

现在切点是在一个地方定义的，并且被多个通知元素所引用。<aop:pointcut>元素定义了一个id为performance的切点。同时修改所有的通知元素，用pointcut-ref属性来引用这个命名切点。

正如程序清单4.10所展示的，<aop:pointcut>元素所定义的切点可以被同一个<aop:aspect>元素之内的所有通知元素引用。如果想让定义的切点能够在多个切面使用，我们可以把<aop:pointcut>元素放在<aop:config>元素的范围内。

4.4.2 声明环绕通知

目前Audience的实现工作得非常棒，但是前置通知和后置通知有一些限制。具体来说，如果不使用成员变量存储信息的话，在前置通知和后置通知之间共享信息非常麻烦。

例如，假设除了进场关闭手机和表演结束后鼓掌，我们还希望观众确保一直关注演出，并报告每个参赛者表演了多长时间。使用前置通知和后置通知实现该功能的唯一方式是在前置通知中记录开始时间并在某个后置通知中报告表演耗费的时间。但这样的话我们必须在一个成员变量中保存开始时间。因为Audience是单例的，如果像这样保存状态的话，将会存在线程安全问题。

相对于前置通知和后置通知，环绕通知在这点上有明显的优势。使用环绕通知，我们可以完成前置通知和后置通知所实现的相同功能，而且只需要在一个方法中 实现。因为整个通知逻辑是在一个方法内实现的，所以不需要使用成员变量保存　状态。

例如，考虑程序清单4.11中新Audience类的watchPerformance()方法，它没有使用任何的注解。

程序清单4.11 **watchPerformance()**方法提供了**AOP**环绕通知

```
package concert;

import org.aspectj.lang.ProceedingJoinPoint;

public class Audience {

  public void watchPerformance(ProceedingJoinPoint jp) {
    try {

      System.out.println("Silencing cell phones");      表演之前

      System.out.println("Taking seats");
      jp.proceed();                  <-- 执行被通知的方法

      System.out.println("CLAP CLAP CLAP!!!");       <-- 表演成功之后
    } catch (Throwable e) {

      System.out.println("Demanding a refund");     <-- 表演失败之后

    }
  }

}
```

在观众切面中，watchPerformance()方法包含了之前四个通知方法的所有功能。不过，所有的功能都放在了这一个方法中，因此这个方法还要负责自身的异常处理。

声明环绕通知与声明其他类型的通知并没有太大区别。我们所需要做的仅仅是使用<aop:around>元素。

程序清单4.12 在**XML**中使用**<aop:around>**元素声明环绕通知

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut
        id="performance"
        expression="execution(** concert.Performance.perform(..))" />

    <aop:around                      <-- 声明环绕通知
        pointcut-ref="performance"
        method="watchPerformance"/>
  </aop:aspect>
</aop:config>
```

像其他通知的XML元素一样，<aop:around>指定了一个切点和一个通知方法的名字。在这里，我们使用跟之前一样的切点，但是为该切点所设置的method属性值为watchPerformance()方法。

4.4.3 为通知传递参数

在4.3.3小节中，我们使用@AspectJ注解创建了一个切面，这个切面

能够记录CompactDisc上每个磁道播放的次数。现在，我们使用XML来配置切面，那就看一下如何完成这一相同的任务。

首先，我们要移除掉TrackCounter上所有的@AspectJ注解。

程序清单**4.13**　无注解的**TrackCounter**

```
package soundsystem;
import java.util.HashMap;
import java.util.Map;

public class TrackCounter {

  private Map<Integer, Integer> trackCounts =
      new HashMap<Integer, Integer>();

  public void countTrack(int trackNumber) {       <-- 要声明为前置通知的方法
    int currentCount = getPlayCount(trackNumber);

    trackCounts.put(trackNumber, currentCount + 1);
  }

  public int getPlayCount(int trackNumber) {
    return trackCounts.containsKey(trackNumber)
        ? trackCounts.get(trackNumber) : 0;
  }

}
```

去掉@AspectJ注解后，TrackCounter显得有些单薄了。现在，除非显式调用countTrack()方法，否则TrackCounter不会记录磁道播放的数量。但是，借助一点Spring XML配置，我们能够让TrackCounter重新变为切面。

如下的程序清单展现了完整的Spring配置，在这个配置中声明了TrackCounter bean和BlankDisc bean，并将TrackCounter转化为切面。

程序清单**4.14**　在**XML**中将**TrackCounter**配置为参数化的切面

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation=
    "http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="trackCounter"
        class="soundsystem.TrackCounter" />          <-- TrackCounter bean

  <bean id="cd"
        class="soundsystem.BlankDisc">          <-- BlankDisc bean
    <property name="title"
              value="Sgt. Pepper's Lonely Hearts Club Band" />
    <property name="artist" value="The Beatles" />
    <property name="tracks">
      <list>
        <value>Sgt. Pepper's Lonely Hearts Club Band</value>
        <value>With a Little Help from My Friends</value>
        <value>Lucy in the Sky with Diamonds</value>
        <value>Getting Better</value>
        <value>Fixing a Hole</value>
        <!-- ...other tracks omitted for brevity... -->
      </list>
    </property>
  </bean>

  <aop:config>
    <aop:aspect ref="trackCounter">          <-- 将 TrackCounter 声明为切面
      <aop:pointcut id="trackPlayed" expression=
          "execution(* soundsystem.CompactDisc.playTrack(int))
              and args(trackNumber)" />

      <aop:before
          pointcut-ref="trackPlayed"
          method="countTrack"/>
    </aop:aspect>
  </aop:config>
</beans>
```

可以看到，我们使用了和前面相同的aop命名空间XML元素，它们会将POJO声明为切面。唯一明显的差别在于切点表达式中包含了一个参数，这个参数会传递到通知方法中。如果你将这个表达式与程序清单4.6中的表达式进行对比会发现它们几乎是相同的。唯一的差别在于这里使用and关键字而不是“&&”（因为在XML中，“&”符号会被解析为实体的开始）。

我们通过练习已经使用Spring的aop命名空间声明了几个基本的切面，那么现在让我们看一下如何使用aop命名空间声明引入切面。

4.4.4 通过切面引入新的功能

在前面的4.3.4小节中，我向你展现了如何借助AspectJ的@DeclareParents注解为被通知的方法神奇地引入新的方法。但是AOP引入并不是AspectJ特有的。使用Spring aop命名空间中的

<aop:declare-parents>元素，我们可以实现相同的功能。

如下的XML代码片段与之前基于AspectJ的引入功能是相同：

```
<aop:aspect>
  <aop:declare-parents
    types-matching="concert.Performance+"
    implement-interface="concert.Encoreable"
    default-impl="concert.DefaultEncoreable"
    />
</aop:aspect>
```

顾名思义，<aop:declare-parents>声明了此切面所通知的bean要在它的对象层次结构中拥有新的父类型。具体到本例中，类型匹配Performance接口（由types-matching属性指定）的那些bean在父类结构中会增加Encoreable接口（由implement-interface属性指定）。最后要解决的问题是Encoreable接口中的方法实现要来自于何处。

这里有两种方式标识所引入接口的实现。在本例中，我们使用default-impl属性用全限定类名来显式指定Encoreable的实现。或者，我们还可以使用delegate-ref属性来标识。

```
<aop:aspect>
  <aop:declare-parents
    types-matching="concert.Performance+"
    implement-interface="concert.Encoreable"
    delegate-ref="encoreableDelegate"
    />
</aop:aspect>
```

delegate-ref属性引用了一个Spring bean作为引入的委托。这需要在Spring上下文中存在一个ID为encoreableDelegate的bean。

```
<bean id="encoreableDelegate"
      class="concert.DefaultEncoreable" />
```

使用default-impl来直接标识委托和间接使用delegate-ref的区别在于后者是Spring bean，它本身可以被注入、通知或使用其他的Spring配置。

4.5　注入AspectJ切面

虽然Spring AOP能够满足许多应用的切面需求，但是与AspectJ相比，Spring AOP 是一个功能比较弱的AOP解决方案。AspectJ提供了Spring AOP所不能支持的许多类型的切点。

例如，当我们需要在创建对象时应用通知，构造器切点就非常方便。不像某些其他面向对象语言中的构造器，Java构造器不同于其他的正常方法。这使得Spring基于代理的AOP无法把通知应用于对象的创建过程。

对于大部分功能来讲，AspectJ切面与Spring是相互独立的。虽然它们可以织入到任意的Java应用中，这也包括了Spring应用，但是在应用AspectJ切面时几乎不会涉及到Spring。

但是精心设计且有意义的切面很可能依赖其他类来完成它们的工作。如果在执行通知时，切面依赖于一个或多个类，我们可以在切面内部实例化这些协作的对象。但更好的方式是，我们可以借助Spring的依赖注入把bean装配进AspectJ切面中。

为了演示，我们为上面的演出创建一个新切面。具体来讲，我们以切面的方式创建一个评论员的角色，他会观看演出并且会在演出之后提供一些批评意见。下面的CriticAspect就是一个这样的切面。

程序清单**4.15**　使用**AspectJ**实现表演的评论员

```
package concert;
public aspect CriticAspect {
  public CriticAspect() {}

  pointcut performance() : execution(* perform(..));

  afterReturning() : performance() {
    System.out.println(criticismEngine.getCriticism());
  }                                                    注入 CriticismEngine

  private CriticismEngine criticismEngine;

  public void setCriticismEngine(CriticismEngine criticismEngine) {   <-
    this.criticismEngine = criticismEngine;
  }
```

CriticAspect的主要职责是在表演结束后为表演发表评论。程序清单4.15中的performance()切点匹配perform()方法。当它与afterReturning()通知一起配合使用时，我们可以让该切面在表演结束时起作用。

程序清单4.15有趣的地方在于并不是评论员自己发表评论，实际上，CriticAspect与一个CriticismEngine对象相协作，在表演结束时，调用该对象的getCriticism()方法来发表一个苛刻的评论。为了避免CriticAspect和CriticismEngine之间产生不必要的耦合，我们通过Setter依赖注入为CriticAspect设置CriticismEngine。图4.9展示了此关系。

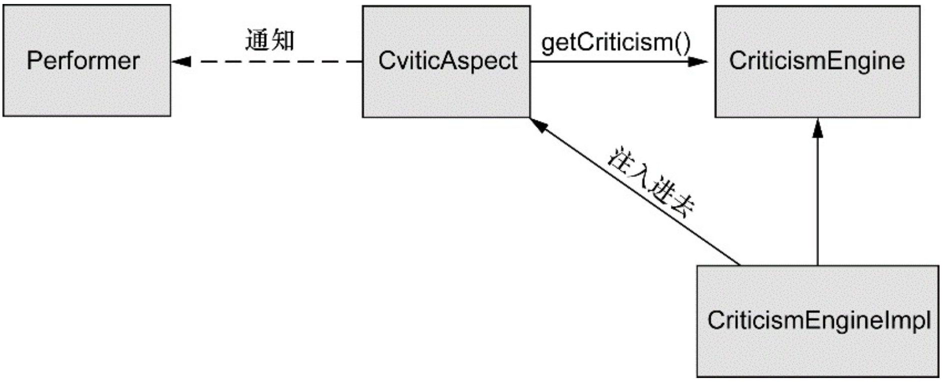

图4.9　切面也需要注入。像其他的bean一样，
Spring可以为AspectJ切面注入依赖

CriticismEngine自身是声明了一个简单getCriticism()方法的接口。程序清单4.16为CriticismEngine的实现。

程序清单**4.16**　要注入到**CriticAspect**中的**CriticismEngine**实现

```
package com.springinaction.springidol;
public class CriticismEngineImpl implements CriticismEngine {
  public CriticismEngineImpl() {}

  public String getCriticism() {
    int i = (int) (Math.random() * criticismPool.length);
    return criticismPool[i];
  }

  // injected
  private String[] criticismPool;
  public void setCriticismPool(String[] criticismPool) {
    this.criticismPool = criticismPool;
  }
}
```

CriticismEngineImpl实现了CriticismEngine接口，通过从注入的评论池中随机选择一个苛刻的评论。这个类可以使用如下的XML声明为一个Spring bean。

```
<bean id="criticismEngine"
    class="com.springinaction.springidol.CriticismEngineImpl">
  <property name="criticisms">
    <list>
      <value>Worst performance ever!</value>
      <value>I laughed, I cried, then I realized I was at the
             wrong show.</value>
      <value>A must see show!</value>
    </list>
  </property>
</bean>
```

到目前为止，一切顺利。我们现在有了一个要赋予CriticAspect的Criticism-Engine实现。剩下的就是为CriticAspect装配CriticismEngineImple。

在展示如何实现注入之前，我们必须清楚AspectJ切面根本不需要Spring就可以织入到我们的应用中。如果想使用Spring的依赖注入为AspectJ切面注入协作者，那我们就需要在Spring配置中把切面声明为一个Spring配置中的<bean>。如下的<bean>声明会把criticismEnginebean注入到CriticAspect中：

```
<bean class="com.springinaction.springidol.CriticAspect"
    factory-method="aspectOf">
  <property name="criticismEngine" ref="criticismEngine" />
</bean>
```

很大程度上，<bean>的声明与我们在Spring中所看到的其他<bean>配置并没有太多的区别，但是最大的不同在于使用了factory-method属性。通常情况下，Spring bean由Spring容器初始化，但是AspectJ切面是由AspectJ在运行期创建的。等到Spring有机会为CriticAspect注入CriticismEngine时，CriticAspect已经被实例化了。

因为Spring不能负责创建CriticAspect，那就不能在 Spring中简单地把CriticAspect声明为一个bean。相反，我们需要一种方式为Spring获得已经由AspectJ创建的CriticAspect实例的句柄，从而可以注入CriticismEngine。幸好，所有的AspectJ切面都提供了一个静态的aspectOf()方法，该方法返回切面的一个单例。所以为了获得切面的实例，我们必须使用factory-method来调用asepctOf()方法而不是调用CriticAspect的构造器方法。

简而言之，Spring不能像之前那样使用<bean>声明来创建一个CriticAspect实例——它已经在运行时由AspectJ创建完成了。Spring需要通过aspectOf()工厂方法获得切面的引用，然后像<bean>元素规定的那样在该对象上执行依赖注入。

4.6 小结

AOP是面向对象编程的一个强大补充。通过AspectJ，我们现在可以把之前分散在应用各处的行为放入可重用的模块中。我们显示地声明在何处如何应用该行为。这有效减少了代码冗余，并让我们的类关注自身的主要功能。

Spring提供了一个AOP框架，让我们把切面插入到方法执行的周围。现在我们已经学会如何把通知织入前置、后置和环绕方法的调用中，以及为处理异常增加自定义的行为。

关于在Spring应用中如何使用切面，我们可以有多种选择。通过使用`@AspectJ`注解和简化的配置命名空间，在Spring中装配通知和切点变得非常简单。

最后，当Spring AOP不能满足需求时，我们必须转向更为强大的AspectJ。对于这些场景，我们了解了如何使用Spring为AspectJ切面注入依赖。

此时此刻，我们已经覆盖了Spring框架的基础知识，了解到如何配置Spring容器以及如何为Spring管理的对象应用切面。正如我们所看到的，这些核心技术为创建松散耦合的应用奠定了坚实的基础。

现在，我们越过这些基础的内容，看一下如何使用Spring构建真实的应用。从下一章开始，首先看到的是如何使用Spring构建Web应用。

---

[1]对应的英文单词词根为encore，指的是演唱会演出结束后应观众要求进行返场表演。——译者注

第2部分　Web中的Spring

Spring通常用来开发Web应用。因此，在第2部分中，将会看到如何使用Spring的MVC框架为应用程序添加Web前端。

在第5章“构建Spring Web应用”中，你将会学习到Spring MVC的基本用法，它是构建在Spring理念之上的一个Web框架。我们将会看到如何编写处理Web请求的控制器以及如何透明地绑定请求参数和负载到业务对象上，同时它还提供了数据检验和错误处理的功能。

在第6章“渲染Web视图”中，将会基于第5章的内容继续讲解，展现了如何得到Spring MVC控制器所生成的模型数据，并将其渲染为用户浏览器中的HTML。这一章的讨论包括JavaServer Pages（JSP）、Apache Tiles和Thymeleaf模板。

在第7章“Spring MVC的高级技术”中，将会学习到构建Web应用时的一些高级技术，包括自定义Spring MVC配置、处理multipart文件上传、处理异常以及使用flash属性跨请求传递数据。

第8章，“使用Spring Web Flow”将会为你展示如何使用Spring Web Flow来构建会话式、基于流程的Web应用程序。

鉴于安全是很多应用程序的重要关注点，因此第9章“保护Web应用”将会为你介绍如何使用Spring Security来为Web应用程序提供安全性，保护应用中的信息。

第5章　构建Spring Web应用程序

本章内容：

- 映射请求到Spring控制器
- 透明地绑定表单参数
- 校验表单提交

作为企业级Java开发者，你可能开发过一些基于Web的应用程序。对于很多Java开发人员来说，基于Web的应用程序是他们主要的关注点。如果你有这方面经验的话，你会意识到这种系统所面临的挑战。具体来讲，状态管理、工作流以及验证都是需要解决的重要特性。HTTP协议的无状态性决定了这些问题都不那么容易解决。

Spring的Web框架就是为了帮你解决这些关注点而设计的。Spring MVC基于模型-视图-控制器（Model-View-Controller，MVC）模式实现，它能够帮你构建像Spring框架那样灵活和松耦合的Web应用程序。

在本章中，我们将会介绍Spring MVC Web框架，并使用新的Spring MVC注解来构建处理各种Web请求、参数和表单输入的控制器。在深入介绍Spring MVC之前，让我们先总体上介绍一下Spring MVC，并建立起Spring MVC运行的基本　配置。

5.1　Spring MVC起步

你见到过孩子们的捕鼠器游戏吗？这真是一个疯狂的游戏，它的目标是发送一个小钢球，让它经过一系列稀奇古怪的装置，最后触发捕鼠器。小钢球穿过各种复杂的配件，从一个斜坡上滚下来，被跷跷板弹起，绕过一个微型摩天轮，然后被橡胶靴从桶中踢出去。经过这些后，小钢球会对那只可怜又无辜的橡胶老鼠进行捕获。

乍看上去，你会认为Spring MVC框架与捕鼠器有些类似。Spring将请求在调度Servlet、处理器映射（handler mapping）、控制器以及视图解析器（view resolver）之间移动，而捕鼠器中的钢球则会在各种斜坡、跷跷板以及摩天轮之间滚动。但是，不要将Spring MVC与Rube Goldberg-esque捕鼠器游戏做过多比较。每一个Spring MVC中的组件

都有特定的目的，并且它也没有那么复杂。

让我们看一下请求是如何从客户端发起，经过Spring MVC中的组件，最终再返回到客户端的。

5.1.1　跟踪Spring MVC的请求

每当用户在Web浏览器中点击链接或提交表单的时候，请求就开始工作了。对请求的工作描述就像是快递投送员。与邮局投递员或FedEx投送员一样，请求会将信息从一个地方带到另一个地方。

请求是一个十分繁忙的家伙。从离开浏览器开始到获取响应返回，它会经历好多站，在每站都会留下一些信息同时也会带上其他信息。图5.1展示了请求使用Spring MVC所经历的所有站点。

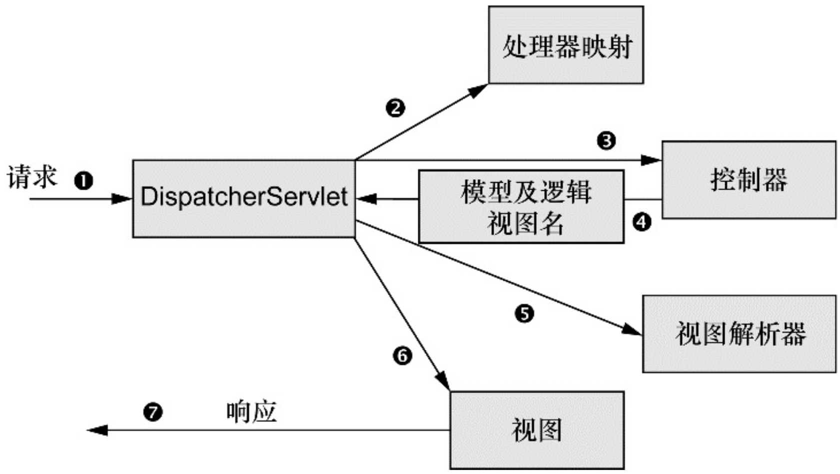

图5.1　一路上请求会将信息带到很多站点，并生产期望的结果

在请求离开浏览器时❶，会带有用户所请求内容的信息，至少会包含请求的URL。但是还可能带有其他的信息，例如用户提交的表单信息。

请求旅程的第一站是Spring的`DispatcherServlet`。与大多数基于Java的Web框架一样，Spring MVC所有的请求都会通过一个前端控制器（front controller）Servlet。前端控制器是常用的Web应用程序模式，在这里一个单实例的Servlet将请求委托给应用程序的其他组件来执行实际的处理。在Spring MVC中，`DispatcherServlet`就是前端控制器。

`DispatcherServlet`的任务是将请求发送给Spring MVC控制器（controller）。控制器是一个用于处理请求的Spring组件。在典型的

应用程序中可能会有多个控制器，DispatcherServlet需要知道应该将请求发送给哪个控制器。所以DispatcherServlet以会查询一个或多个处理器映射（handler mapping）❷来确定请求的下一站在哪里。处理器映射会根据请求所携带的URL信息来进行决策。

一旦选择了合适的控制器，DispatcherServlet会将请求发送给选中的控制器❸。到了控制器，请求会卸下其负载（用户提交的信息）并耐心等待控制器处理这些信息。（实际上，设计良好的控制器本身只处理很少甚至不处理工作，而是将业务逻辑委托给一个或多个服务对象进行处理。）

控制器在完成逻辑处理后，通常会产生一些信息，这些信息需要返回给用户并在浏览器上显示。这些信息被称为模型（model）。不过仅仅给用户返回原始的信息是不够的——这些信息需要以用户友好的方式进行格式化，一般会是HTML。所以，信息需要发送给一个视图（view），通常会是JSP。

控制器所做的最后一件事就是将模型数据打包，并且标示出用于渲染输出的视图名。它接下来会将请求连同模型和视图名发送回DispatcherServlet❹。

这样，控制器就不会与特定的视图相耦合，传递给DispatcherServlet的视图名并不直接表示某个特定的JSP。实际上，它甚至并不能确定视图就是JSP。相反，它仅仅传递了一个逻辑名称，这个名字将会用来查找产生结果的真正视图。DispatcherServlet将会使用视图解析器（view resolver）❺来将逻辑视图名匹配为一个特定的视图实现，它可能是也可能不是JSP。

既然DispatcherServlet已经知道由哪个视图渲染结果，那请求的任务基本上也就完成了。它的最后一站是视图的实现（可能是JSP）❻，在这里它交付模型数据。请求的任务就完成了。视图将使用模型数据渲染输出，这个输出会通过响应对象传递给客户端（不会像听上去那样硬编码）❼。

可以看到，请求要经过很多的步骤，最终才能形成返回给客户端的响应。大多数的步骤都是在Spring框架内部完成的，也就是图5.1所示的组件中。尽管本章的主要内容都关注于如何编写控制器，但在此之前我们首先看一下如何搭建Spring MVC的基础组件。

5.1.2 搭建Spring MVC

基于图5.1，看上去我们需要配置很多的组成部分。幸好，借助于最近几个Spring新版本的功能增强，开始使用Spring MVC变得非常简单了。现在，我们要使用最简单的方式来配置Spring MVC：所要实现的功能仅限于运行我们所创建的控制器。在第7章中，我们会看一些其他的配置选项。

配置DispatcherServlet

DispatcherServlet是Spring MVC的核心。在这里请求会第一次接触到框架，它要负责将请求路由到其他的组件之中。

按照传统的方式，像DispatcherServlet这样的Servlet会配置在web.xml文件中，这个文件会放到应用的WAR包里面。当然，这是配置DispatcherServlet的方法之一。但是，借助于Servlet 3规范和Spring 3.1的功能增强，这种方式已经不是唯一的方案了，这也不是我们本章所使用的配置方法。

我们会使用Java将DispatcherServlet配置在Servlet容器中，而不会再使用web.xml文件。如下的程序清单展示了所需的Java类。

程序清单**5.1**　配置**DispatcherServlet**

```
package spittr.config;
import org.springframework.web.servlet.support.
            AbstractAnnotationConfigDispatcherServletInitializer;

public class SpittrWebAppInitializer
      extends AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected String[] getServletMappings() {        <—— 将 DispatcherServlet 映射到“/”
    return new String[] { "/" };
  }

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] { RootConfig.class };
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {   <—— 指定配置类
    return new Class<?>[] { WebConfig.class };
  }

}
```

在我们深入介绍程序清单5.1之前，你可能想知道spittr到底是什么意思。这个类的名字是SpittrWebAppInitializer，它位于名为spittr.config的包中。我稍后会对其进行介绍（在5.1.3小节中），但现在，你只需要知道我们所要创建的应用名为Spittr。

要理解程序清单5.1是如何工作的，我们可能只需要知道扩展AbstractAnnotation-ConfigDispatcherServletInitializer的任意类都会自动地配置Dispatcher-Servlet和Spring应用上下文，Spring的应用上下文会位于应用程序的Servlet上下文之中。

AbstractAnnotationConfigDispatcherServletInitializer剖析

如果你坚持要了解更多细节的话，那就看这里吧。在Servlet 3.0环境中，容器会在类路径中查找实现javax.servlet.ServletContainerInitializer接口的类，如果能发现的话，就会用它来配置Servlet容器。

Spring提供了这个接口的实现，名为SpringServletContainerInitializer，这个类反过来又会查找实现WebApplicationInitializer的类并将配置的任务交给它们来完成。Spring 3.2引入了一个便利的WebApplicationInitializer基础实现，也就是AbstractAnnotationConfigDispatcherServletInitiali因为我们的Spittr-WebAppInitializer扩展了AbstractAnnotationConfig DispatcherServlet-Initializer（同时也就实现了WebApplicationInitializer），因此当部署到Servlet 3.0容器中的时候，容器会自动发现它，并用它来配置Servlet上下文。

尽管它的名字很长，但是AbstractAnnotationConfigDispatcherServlet-Initializer使用起来很简便。在程序清单5.1中，SpittrWebAppInitializer重写了三个方法。

第一个方法是getServletMappings()，它会将一个或多个路径映射到DispatcherServlet上。在本例中，它映射的是“/”，这表示它会是应用的默认Servlet。它会处理进入应用的所有请求。

为了理解其他的两个方法，我们首先要理解DispatcherServlet和一个Servlet监听器（也就是ContextLoaderListener）的关系。

两个应用上下文之间的故事

当DispatcherServlet启动的时候，它会创建Spring应用上下文，

并加载配置文件或配置类中所声明的bean。在程序清单5.1的getServletConfigClasses()方法中，我们要求DispatcherServlet加载应用上下文时，使用定义在WebConfig配置类（使用Java配置）中的bean。

但是在Spring Web应用中，通常还会有另外一个应用上下文。另外的这个应用上下文是由ContextLoaderListener创建的。

我们希望DispatcherServlet加载包含Web组件的bean，如控制器、视图解析器以及处理器映射，而ContextLoaderListener要加载应用中的其他bean。这些bean通常是驱动应用后端的中间层和数据层组件。

实际上，AbstractAnnotationConfigDispatcherServletInitia会同时创建DispatcherServlet和ContextLoaderListener。GetServlet-ConfigClasses()方法返回的带有@Configuration注解的类将会用来定义DispatcherServlet应用上下文中的bean。getRootConfigClasses()方法返回的带有@Configuration注解的类将会用来配置ContextLoaderListener创建的应用上下文中的bean。

在本例中，根配置定义在RootConfig中，DispatcherServlet的配置声明在WebConfig中。稍后我们将会看到这两个类的内容。

需要注意的是，通过AbstractAnnotationConfigDispatcherServlet-Initializer来配置DispatcherServlet是传统web.xml方式的替代方案。如果你愿意的话，可以同时包含web.xml和AbstractAnnotationConfigDispatcher-ServletInitializer，但这其实并没有必要。

如果按照这种方式配置DispatcherServlet，而不是使用web.xml的话，那唯一问题在于它只能部署到支持Servlet 3.0的服务器中才能正常工作，如Tomcat 7或更高版本。Servlet 3.0规范在2009年12月份就发布了，因此很有可能你会将应用部署到支持Servlet 3.0的Servlet容器之中。

如果你还没有使用支持Servlet 3.0的服务器，那么在AbstractAnnotation-

ConfigDispatcherServletInitializer子类中配置DispatcherServlet的方法就不适合你了。你别无选择，只能使用web.xml了。我们将会在第7章学习web.xml和其他配置选项。但现在，我们先看一下程序清单5.1中所引用的WebConfig和RootConfig，了解一下如何启用Spring MVC。

启用Spring MVC

我们有多种方式来配置DispatcherServlet，与之类似，启用Spring MVC组件的方法也不仅一种。以前，Spring是使用XML进行配置的，你可以使用<mvc:annotation-driven>启用注解驱动的Spring MVC。

我们会在第7章讨论Spring MVC配置可选项的时候，再讨论<mvc:annotation-driven>。不过，现在我们会让Spring MVC的搭建过程尽可能简单并基于Java进行配置。

我们所能创建的最简单的Spring MVC配置就是一个带有@EnableWebMvc注解的类：

```
package spittr.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@EnableWebMvc
public class WebConfig {
}
```

这可以运行起来，它的确能够启用Spring MVC，但还有不少问题要解决：

- 没有配置视图解析器。如果这样的话，Spring默认会使用BeanNameView-Resolver，这个视图解析器会查找ID与视图名称匹配的bean，并且查找的bean要实现View接口，它以这样的方式来解析视图。
- 没有启用组件扫描。这样的结果就是，Spring只能找到显式声明在配置类中的控制器。
- 这样配置的话，DispatcherServlet会映射为应用的默认Servlet，所以它会处理所有的请求，包括对静态资源的请求，如图片和样式表（在大多数情况下，这可能并不是你想要的效果）。

因此，我们需要在WebConfig这个最小的Spring MVC配置上再加一些内容，从而让它变得真正有用。如下程序清单中的WebConfig解

决了上面所述的问题。

程序清单**5.2**　最小但可用的**Spring MVC**配置

```
package spittr.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.
                                  DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.
                                  WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.
                                     InternalResourceViewResolver;

@Configuration
@EnableWebMvc                                   <-- 启用 Spring MVC
@ComponentScan("spitter.web")                               <-- 启用组件扫描
public class WebConfig
       extends WebMvcConfigurerAdapter {

  @Bean
  public ViewResolver viewResolver() {                        配置 JSP 视图解析器
    InternalResourceViewResolver resolver =                <--
            new InternalResourceViewResolver();
    resolver.setPrefix("/WEB-INF/views/");
    resolver.setSuffix(".jsp");
    resolver.setExposeContextBeansAsAttributes(true);
    return resolver;
  }

  @Override                                                     配置静态资源的处理
  public void configureDefaultServletHandling(                <--
        DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
  }

}
```

在程序清单5.2中第一件需要注意的事情是`WebConfig`现在添加了`@Component-Scan`注解，因此将会扫描spitter.web包来查找组件。稍后你就会看到，我们所编写的控制器将会带有`@Controller`注解，这会使其成为组件扫描时的候选bean。因此，我们不需要在配置类中显式声明任何的控制器。

接下来，我们添加了一个`ViewResolver` bean。更具体来讲，是`Internal-ResourceViewResolver`。我们将会在第6章更为详细地讨论视图解析器。我们只需要知道它会查找JSP文件，在查找的时候，它会在视图名称上加一个特定的前缀和后缀（例如，名为`home`的视图将会解析为/WEB-INF/views/home.jsp）。

最后，新的`WebConfig`类还扩展了`WebMvcConfigurerAdapter`并重写了其`configureDefaultServletHandling()`方法。通过调用`DefaultServlet-HandlerConfigurer`的`enable()`方法，

我们要求DispatcherServlet将对静态资源的请求转发到Servlet容器中默认的Servlet上，而不是使用DispatcherServlet本身来处理此类请求。

WebConfig已经就绪，那RootConfig呢？因为本章聚焦于Web开发，而Web相关的配置通过DispatcherServlet创建的应用上下文都已经配置好了，因此现在的RootConfig相对很简单：

```
package spittr.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.ComponentScan.Filter;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@ComponentScan(basePackages={"spitter"},
    excludeFilters={
        @Filter(type=FilterType.ANNOTATION, value=EnableWebMvc.class)
    })
public class RootConfig {
}
```

唯一需要注意的是RootConfig使用了@ComponentScan注解。这样的话，在本书中，我们就有很多机会用非Web的组件来充实完善RootConfig。

现在，我们基本上已经可以开始使用Spring MVC构建Web应用了。此时，最大的问题在于，我们要构建的应用到底是什么。

5.1.3 Spittr应用简介

为了实现在线社交的功能，我们将要构建一个简单的微博（microblogging）应用。在很多方面，我们所构建的应用与最早的微博应用Twitter很类似。在这个过程中，我们会添加一些小的变化。当然，我们要使用Spring技术来构建这个应用。

因为从Twitter借鉴了灵感并且通过Spring来进行实现，所以它就有了一个名字：Spitter。再进一步，应用网站命名中流行的模式，如Flickr，我们去掉字母e，这样的话，我们就将这个应用称为Spittr。这个名称也有助于区分应用名称和领域类型，因为我们将会创建一个名为Spitter的领域类。

Spittr应用有两个基本的领域概念：Spitter（应用的用户）和Spittle（用户发布的简短状态更新）。当我们在书中完善Spittr应用的功能时，将会介绍这两个领域概念。在本章中，我们会构建应用的Web层，创建展现Spittle的控制器以及处理用户注册成为Spitter的表

单。

舞台已经搭建完成了。我们已经配置了DispatcherServlet，启用了基本的Spring MVC组件并确定了目标应用。让我们进入本章的核心内容：使用Spring MVC控制器处理Web请求。

5.2 编写基本的控制器

在Spring MVC中，控制器只是方法上添加了@RequestMapping注解的类，这个注解声明了它们所要处理的请求。

开始的时候，我们尽可能简单，假设控制器类要处理对“/”的请求，并渲染应用的首页。程序清单5.3所示的HomeController可能是最简单的Spring MVC控制器类了。

程序清单**5.3 HomeController**：超级简单的控制器

```
package spittr.web;
import static org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller                          <— 声明为一个控制器
public class HomeController {

  @RequestMapping(value="/", method=GET)        <— 处理对“/”的 GET 请求
  public String home() {
    return "home";                  <— 视图名为 home
  }

}
```

你可能注意到的第一件事情就是HomeController带有@Controller注解。很显然这个注解是用来声明控制器的，但实际上这个注解对Spring MVC本身的影响并不大。

HomeController是一个构造型（stereotype）的注解，它基于@Component注解。在这里，它的目的就是辅助实现组件扫描。因为HomeController带有@Controller注解，因此组件扫描器会自动找到HomeController，并将其声明为Spring应用上下文中的一个bean。

其实，你也可以让HomeController带有@Component注解，它所实现的效果是一样的，但是在表意性上可能会差一些，无法确定HomeController是什么组件类型。

HomeController唯一的一个方法，也就是home()方法，带有@RequestMapping注解。它的value属性指定了这个方法所要处理的请求路径，method属性细化了它所处理的HTTP方法。在本例中，当收到对“/”的HTTP GET请求时，就会调用home()方法。

你可以看到，home()方法其实并没有做太多的事情：它返回了一个String类型的“home”。这个String将会被Spring MVC解读为要渲染的视图名称。DispatcherServlet会要求视图解析器将这个逻辑名称解析为实际的视图。

鉴于我们配置InternalResourceViewResolver的方式，视图名“home”将会解析为“/WEB-INF/views/home.jsp”路径的JSP。现在，我们会让Spittr应用的首页相当简单，如下所示。

程序清单**5.4　Spittr**应用的首页，定义为一个简单的**JSP**

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
  <head>
    <title>Spittr</title>
    <link rel="stylesheet"
          type="text/css"
          href="<c:url value="/resources/style.css" />" >
  </head>
  <body>
    <h1>Welcome to Spittr</h1>

    <a href="<c:url value="/spittles" />">Spittles</a> |
    <a href="<c:url value="/spitter/register" />">Register</a>
  </body>
</html>
```

这个JSP并没有太多需要注意的地方。它只是欢迎应用的用户，并提供了两个链接：一个是查看Spittle列表，另一个是在应用中进行注册。图5.2展现了此时的首页是什么样子的。

在本章完成之前，我们将会实现处理这些请求的控制器方法。但现在，让我们对这个控制器发起一些请求，看一下它是否能够正常工作。测试控制器最直接的办法可能就是构建并部署应用，然后通过浏览器对其进行访问，但是自动化测试可能会给你更快的反馈和更一致的独立结果。所以，让我们编写一个针对HomeController的测试。

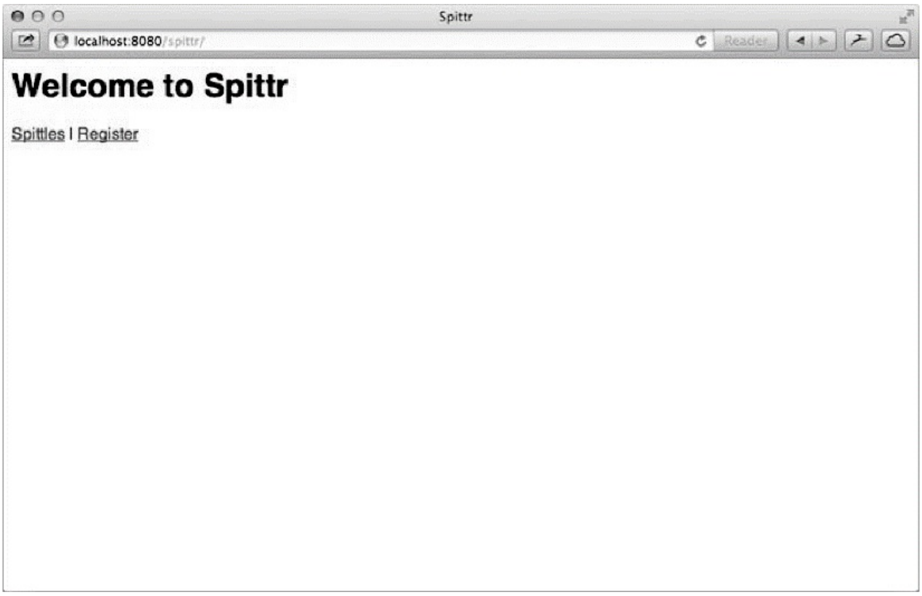

图5.2　当前的Spittr首页

5.2.1　测试控制器

让我们再审视一下HomeController。如果你眼神不太好的话，你甚至可能注意不到这些注解，所看到的仅仅是一个简单的POJO。我们都知道测试POJO是很容易的。因此，我们可以编写一个简单的类来测试HomeController，如下所示：

程序清单5.5　**HomeControllerTest**：测试**HomeController**

```
package spittr.web;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import spittr.web.HomeController;

public class HomeControllerTest {
  @Test
  public void testHomePage() throws Exception {
    HomeController controller = new HomeController();
    assertEquals("home", controller.home());
  }
}
```

程序清单5.5中的测试很简单，但它只测试了home()方法中会发生什么。在测试中会直接调用home()方法，并断言返回包含“home”值的String。它完全没有站在Spring MVC控制器的视角进行测试。这个测试没有断言当接收到针对“/”的GET请求时会调用home()方法。因为它返回的值就是“home”，所以也没有真正判断home是视图的名称。

不过从Spring 3.2开始，我们可以按照控制器的方式来测试Spring MVC中的控制器了，而不仅仅是作为POJO进行测试。Spring现在包含了一

种mock Spring MVC并针对控制器执行HTTP请求的机制。这样的话，在测试控制器的时候，就没有必要再启动Web服务器和Web浏览器了。

为了阐述如何测试Spring MVC的控制器，我们重写HomeControllerTest并使用Spring MVC中新的测试特性。程序清单5.6展现了新的HomeControllerTest。

程序清单**5.6**　改进**HomeControllerTest**

```
package spittr.web;
import static
  org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
  org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
import static
  org.springframework.test.web.servlet.setup.MockMvcBuilders.*;
import org.junit.Test;
import org.springframework.test.web.servlet.MockMvc;
import spittr.web.HomeController;

public class HomeControllerTest {
  @Test
  public void testHomePage() throws Exception {
    HomeController controller = new HomeController();
    MockMvc mockMvc =                                  <— 搭建 MockMvc
        standaloneSetup(controller).build();

    mockMvc.perform(get("/"))                <— 对 “/” 执行 GET 请求
           .andExpect(view().name("home"));            <— 预期得到 home 视图
  }
}
```

尽管新版本的测试只比之前版本多了几行代码，但是它更加完整地测试了HomeController。这次我们不是直接调用home()方法并测试它的返回值，而是发起了对“/”的GET请求，并断言结果视图的名称为home。它首先传递一个HomeController实例到MockMvcBuilders.standaloneSetup()并调用build()来构建MockMvc实例。然后它使用MockMvc实例来执行针对“/”的GET请求并设置期望得到的视图名称。

5.2.2　定义类级别的请求处理

现在，已经为HomeController编写了测试，那么我们可以做一些重构，并通过测试来保证不会对功能造成什么破坏。我们可以做的一件事就是拆分@RequestMapping，并将其路径映射部分放到类级别上。程序清单5.7展示了这个过程。

程序清单**5.7**　拆分**HomeController**中的**@RequestMapping**

```
package spittr.web;
import static org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/")                    <— 将控制器映射到“/”
public class HomeController {

  @RequestMapping(method=GET)            <— 处理 GET 请求
  public String home() {
    return "home";          <— 视图名为 home
  }

}
```

在这个新版本的HomeController中，路径现在被转移到类级别的@RequestMapping上，而HTTP方法依然映射在方法级别上。当控制器在类级别上添加@RequestMapping注解时，这个注解会应用到控制器的所有处理器方法上。处理器方法上的@RequestMapping注解会对类级别上的@RequestMapping的声明进行补充。

就HomeController而言，这里只有一个控制器方法。与类级别的@Request-Mapping合并之后，这个方法的@RequestMapping表明home()将会处理对“/”路径的GET请求。

换言之，我们其实没有改变任何功能，只是将一些代码换了个地方，但是HomeController所做的事情和以前是一样的。因为我们现在有了测试，所以可以确保在这个过程中，没有对原有的功能造成破坏。

当我们在修改@RequestMapping时，还可以对HomeController做另外一个变更。@RequestMapping的value属性能够接受一个String类型的数组。到目前为止，我们给它设置的都是一个String类型的“/”。但是，我们还可以将它映射到对“/homepage”的请求，只需将类级别的@RequestMapping改为如下所示：

```
@Controller
@RequestMapping({"/", "/homepage"})
public class HomeController {
...
}
```

现在，HomeController的home()方法能够映射到对“/”和“/homepage”的GET请求。

5.2.3 传递模型数据到视图中

到现在为止，就编写超级简单的控制器来说，HomeController已经是一个不错的样例了。但是大多数的控制器并不是这么简单。在Spittr应用中，我们需要有一个页面展现最近提交的Spittle列表。因此，我们需要一个新的方法来处理这个页面。

首先，需要定义一个数据访问的Repository。为了实现解耦以及避免陷入数据库访问的细节之中，我们将Repository定义为一个接口，并在稍后实现它（第10章中）。此时，我们只需要一个能够获取Spittle列表的Repository，如下所示的SpittleRepository功能已经足够了：

```
package spittr.data;
import java.util.List;
import spittr.Spittle;

public interface SpittleRepository {
  List<Spittle> findSpittles(long max, int count);
}
```

findSpittles()方法接受两个参数。其中max参数代表所返回的Spittle中，Spittle ID属性的最大值，而count参数表明要返回多少个Spittle对象。为了获得最新的20个Spittle对象，我们可以这样调用findSpittles()：

```
List<Spittle> recent =
        spittleRepository.findSpittles(Long.MAX_VALUE, 20);
```

现在，我们让Spittle类尽可能的简单，如下面的程序清单5.8所示。它的属性包括消息内容、时间戳以及Spittle发布时对应的经纬度。

程序清单**5.8**　**Spittle**类：包含消息内容、时间戳和位置信息

```java
package spittr;
import java.util.Date;

public class Spittle {
  private final Long id;
  private final String message;
  private final Date time;
  private Double latitude;
  private Double longitude;

  public Spittle(String message, Date time) {
    this(message, time, null, null);
  }

  public Spittle(
      String message, Date time, Double longitude, Double latitude) {
    this.id = null;
    this.message = message;
    this.time = time;
    this.longitude = longitude;
    this.latitude = latitude;
  }

  public long getId() {
    return id;
  }

  public String getMessage() {
    return message;
  }
  public Date getTime() {
    return time;
  }

  public Double getLongitude() {
    return longitude;
  }

  public Double getLatitude() {
    return latitude;
  }

  @Override
  public boolean equals(Object that) {
    return EqualsBuilder.reflectionEquals(this, that, "id", "time");
  }

  @Override
  public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, "id", "time");
  }
}
```

就大部分内容来看，`Spittle`就是一个基本的POJO数据对象——没有什么复杂的。唯一要注意的是，我们使用Apache Common Lang包来实现`equals()`和`hashCode()`方法。这些方法除了常规的作用以外，当我们为控制器的处理器方法编写测试时，它们也是有用的。

既然我们说到了测试，那么我们继续讨论这个话题并为新的控制器方法编写测试。如下的程序清单使用Spring的`MockMvc`来断言新的处理器方法中你所期望的行为。

程序清单5.9　测试SpittleController处理针对“/spittles”的GET请求

```
@Test
public void shouldShowRecentSpittles() throws Exception {
  List<Spittle> expectedSpittles = createSpittleList(20);
  SpittleRepository mockRepository =                    <— Mock Repository
      mock(SpittleRepository.class);
  when(mockRepository.findSpittles(Long.MAX_VALUE, 20))
       .thenReturn(expectedSpittles);

  SpittleController controller =
          new SpittleController(mockRepository);

  SpittleController controller =
         new SpittleController(mockRepository);
  MockMvc mockMvc = standaloneSetup(controller)          <— Mock Spring MVC
      .setSingleView(
          new InternalResourceView("/WEB-INF/views/spittles.jsp"))
      .build();

  mockMvc.perform(get("/spittles"))              <— 对“/spittles”发起 GET 请求
    .andExpect(view().name("spittles"))
    .andExpect(model().attributeExists("spittleList"))
    .andExpect(model().attribute("spittleList",          <—断言期望的值
        hasItems(expectedSpittles.toArray())));
}
...

private List<Spittle> createSpittleList(int count) {
  List<Spittle> spittles = new ArrayList<Spittle>();
  for (int i=0; i < count; i++) {
    spittles.add(new Spittle("Spittle " + i, new Date()));
  }
  return spittles;
}
```

这个测试首先会创建SpittleRepository接口的mock实现，这个实现会从它的findSpittles()方法中返回20个Spittle对象。然后，它将这个Repository注入到一个新的SpittleController实例中，然后创建MockMvc并使用这个控制器。

需要注意的是，与HomeController不同，这个测试在MockMvc构造器上调用了setSingleView()。这样的话，mock框架就不用解析控制器中的视图名了。在很多场景中，其实没有必要这样做。但是对于这个控制器方法，视图名与请求路径是非常相似的，这样按照默认的视图解析规则时，MockMvc就会发生失败，因为无法区分视图路径和控制器的路径。在这个测试中，构建InternalResourceView时所设置的实际路径是无关紧要的，但我们将其设置为与InternalResourceViewResolver配置一致。

这个测试对“/spittles”发起GET请求，然后断言视图的名称为spittles并且模型中包含名为spittleList的属性，在spittleList中包含预期的内容。

当然，如果此时运行测试的话，它将会失败。它不是运行失败，而是在编译的时候就会失败。这是因为我们还没有编写SpittleController。现在，我们创建SpittleController，

让它满足程序清单5.9的预期。如下的SpittleController实现将会满足以上测试的要求。

程序清单5.10　**SpittleController**：在模型中放入最新的**spittle**列表

```
package spittr.web;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import spittr.Spittle;
import spittr.data.SpittleRepository;

@Controller
@RequestMapping("/spittles")
public class SpittleController {

  private SpittleRepository spittleRepository;
  @Autowired
  public SpittleController(                <— 注入 SpittleRepository
      SpittleRepository spittleRepository) {
    this.spittleRepository = spittleRepository;
  }

  @RequestMapping(method=RequestMethod.GET)

  public String spittles(Model model) {
    model.addAttribute(                     <— 将 spittle 添加到模型中
        spittleRepository.findSpittles(
            Long.MAX_VALUE, 20));
    return "spittles";                     <— 返回视图名
  }

}
```

我们可以看到SpittleController有一个构造器，这个构造器使用了@Autowired注解，用来注入SpittleRepository。这个SpittleRepository随后又用在spittles()方法中，用来获取最新的spittle列表。

需要注意的是，我们在spittles()方法中给定了一个Model作为参数。这样，spittles()方法就能将Repository中获取到的Spittle列表填充到模型中。Model实际上就是一个Map（也就是key-value对的集合），它会传递给视图，这样数据就能渲染到客户端了。当调用addAttribute()方法并且不指定key的时候，那么key会根据值的对象类型推断确定。在本例中，因为它是一个List<Spittle>，因此，键将会推断为spittleList。

spittles()方法所做的最后一件事是返回spittles作为视图的名字，这个视图会渲染模型。

如果你希望显式声明模型的key的话，那也尽可以进行指定。例如，下面这个版本的spittles()方法与程序清单5.10中的方法作用是一

样的：

```
@RequestMapping(method=RequestMethod.GET)
public String spittles(Model model) {
  model.addAttribute("spittleList",
      spittleRepository.findSpittles(Long.MAX_VALUE, 20));
  return "spittles";
}
```

如果你希望使用非Spring类型的话，那么可以用java.util.Map来代替Model。下面这个版本的spittles()方法与之前的版本在功能上是一样的：

```
@RequestMapping(method=RequestMethod.GET)
public String spittles(Map model) {
  model.put("spittleList",
          spittleRepository.findSpittles(Long.MAX_VALUE, 20));
  return "spittles";
}
```

既然我们现在提到了各种可替代的方案，那下面还有另外一种方式来编写spittles()方法：

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles() {
  return spittleRepository.findSpittles(Long.MAX_VALUE, 20));
}
```

这个版本与其他的版本有些差别。它并没有返回视图名称，也没有显式地设定模型，这个方法返回的是Spittle列表。当处理器方法像这样返回对象或集合时，这个值会放到模型中，模型的key会根据其类型推断得出（在本例中，也就是spittleList）。

而逻辑视图的名称将会根据请求路径推断得出。因为这个方法处理针对“/spittles”的GET请求，因此视图的名称将会是spittles（去掉开头的斜线）。

不管你选择哪种方式来编写spittles()方法，所达成的结果都是相同的。模型中会存储一个Spittle列表，key为spittleList，然后这个列表会发送到名为spittles的视图中。按照我们配置InternalResourceViewResolver的方式，视图的JSP将会是“/WEB-INF/views/spittles.jsp”。

现在，数据已经放到了模型中，在JSP中该如何访问它呢？实际上，当视图是JSP的时候，模型数据会作为请求属性放到请求（request）之中。因此，在spittles.jsp文件中可以使用JSTL（JavaServer Pages Standard Tag Library）的<c:forEach>标签渲染spittle列表：

```
<c:forEach items="${spittleList}" var="spittle" >
  <li id="spittle_<c:out value="spittle.id"/>">
    <div class="spittleMessage">
      <c:out value="${spittle.message}" />
    </div>
    <div>
      <span class="spittleTime"><c:out value="${spittle.time}" /></span>
      <span class="spittleLocation">
          (<c:out value="${spittle.latitude}" />,
          <c:out value="${spittle.longitude}" />)</span>
    </div>
  </li>
</c:forEach>
```

图5.3为显示效果，能够让你对它在Web浏览器中是什么样子有个可视化的印象。

尽管SpittleController很简单，但是它依然比HomeController更进一步了。不过，SpittleController和HomeController都没有处理任何形式的输入。现在，让我们扩展SpittleController，让它从客户端接受一些输入。

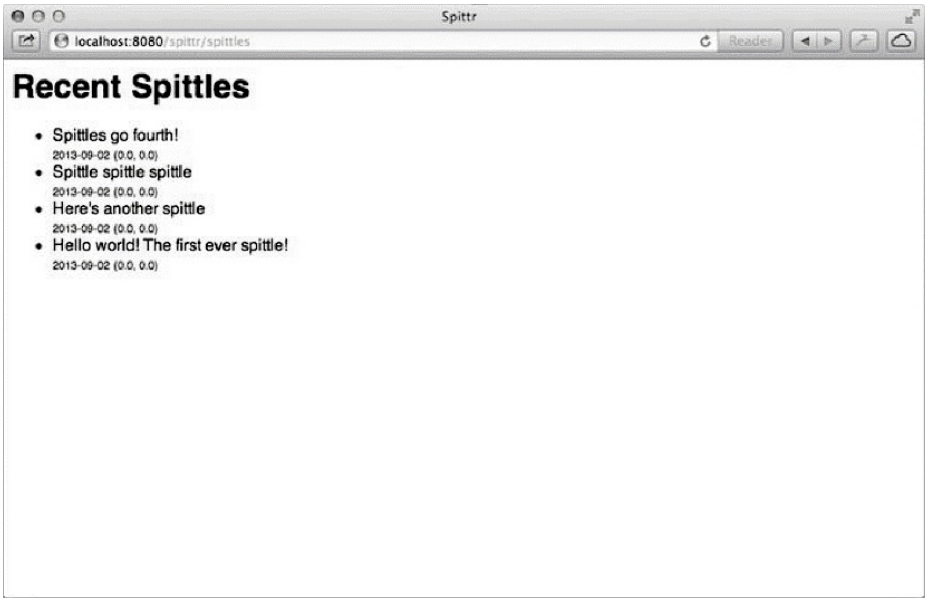

图5.3　控制器中的Spittle模型数据将会作为请求参数，并在Web页面上渲染为列表的形式

5.3　接受请求的输入

有些Web应用是只读的。人们只能通过浏览器在站点上闲逛，阅读服务器发送到浏览器中的内容。

不过，这并不是一成不变的。众多的Web应用允许用户参与进去，将

数据发送回服务器。如果没有这项能力的话，那Web将完全是另一番景象。

Spring MVC允许以多种方式将客户端中的数据传送到控制器的处理器方法中，包括：

- 查询参数（Query Parameter）。
- 表单参数（Form Parameter）。
- 路径变量（Path Variable）。

你将会看到如何编写控制器处理这些不同机制的输入。作为开始，我们先看一下如何处理带有查询参数的请求，这也是客户端往服务器端发送数据时，最简单和最直接的方式。

5.3.1 处理查询参数

在Spittr应用中，我们可能需要处理的一件事就是展现分页的Spittle列表。在现在的`SpittleController`中，它只能展现最新的Spittle，并没有办法向前翻页查看以前编写的Spittle历史记录。如果你想让用户每次都能查看某一页的Spittle历史，那么就需要提供一种方式让用户传递参数进来，进而确定要展现哪些Spittle集合。

在确定该如何实现时，假设我们要查看某一页Spittle列表，这个列表会按照最新的Spittle在前的方式进行排序。因此，下一页中第一条的ID肯定会早于当前页最后一条的ID。所以，为了显示下一页的Spittle，我们需要将一个Spittle的ID传入进来，这个ID要恰好小于当前页最后一条Spittle的ID。另外，你还可以传入一个参数来确定要展现的Spittle数量。

为了实现这个分页的功能，我们所编写的处理器方法要接受如下的参数：

- `before`参数（表明结果中所有`Spittle`的ID均应该在这个值之前）。
- `count`参数（表明在结果中要包含的`Spittle`数量）。

为了实现这个功能，我们将程序清单5.10中的`spittles()`方法替换为使用`before`和`count`参数的新`spittles()`方法。我们首先添加一个测试，这个测试反映了新`spittles()`方法的功能。

程序清单**5.11**　用来测试分页**Spittle**列表的新方法

```
@Test
public void shouldShowPagedSpittles() throws Exception {
  List<Spittle> expectedSpittles = createSpittleList(50);
  SpittleRepository mockRepository = mock(SpittleRepository.class);
  when(mockRepository.findSpittles(238900, 50))        <-- 预期的 max 和 count 参数
      .thenReturn(expectedSpittles);

  SpittleController controller =
      new SpittleController(mockRepository);
  MockMvc mockMvc = standaloneSetup(controller)
      .setSingleView(
          new InternalResourceView("/WEB-INF/views/spittles.jsp"))
      .build();

  mockMvc.perform(get("/spittles?max=238900&count=50"))      <-- 传入 max 和 count 参数
    .andExpect(view().name("spittles"))
    .andExpect(model().attributeExists("spittleList"))
    .andExpect(model().attribute("spittleList",
              hasItems(expectedSpittles.toArray())));
}
```

这个测试方法与程序清单5.9中的测试方法关键区别在于它针对“/spittles”发送GET请求，同时还传入了max和count参数。它测试了这些参数存在时的处理器方法，而另一个测试方法则测试了没有这些参数时的情景。这两个测试就绪后，我们就能确保不管控制器发生什么样的变化，它都能够处理这两种类型的请求：

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles(
    @RequestParam("max") long max,
    @RequestParam("count") int count) {
  return spittleRepository.findSpittles(max, count);
}
```

SpittleController中的处理器方法要同时处理有参数和没有参数的场景，那我们需要对其进行修改，让它能接受参数，同时，如果这些参数在请求中不存在的话，就使用默认值Long.MAX_VALUE和20。@RequestParam注解的defaultValue属性可以完成这项任务：

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles(
    @RequestParam(value="max",
                  defaultValue=MAX_LONG_AS_STRING) long max,
    @RequestParam(value="count", defaultValue="20") int count) {
  return spittleRepository.findSpittles(max, count);
}
```

现在，如果max参数没有指定的话，它将会是Long类型的最大值。因为查询参数都是String类型的，因此defaultValue属性需要String类型的值。因此，使用Long.MAX_VALUE是不行的。我们可以将Long.MAX_VALUE转换为名为MAX_LONG_-AS_STRING的String类型常量：

```
private static final String MAX_LONG_AS_STRING =
        Long.toString(Long.MAX_VALUE);
```

尽管defaultValue属性给定的是String类型的值，但是当绑定到方法的max参数时，它会转换为Long类型。

如果请求中没有count参数的话，count参数的默认值将会设置为20。

请求中的查询参数是往控制器中传递信息的常用手段。另外一种方式也很流行，尤其是在构建面向资源的控制器时，这种方式就是将传递参数作为请求路径的一部分。让我们看一下如何将路径变量作为请求路径的一部分，从而实现信息的输入。

5.3.2 通过路径参数接受输入

假设我们的应用程序需要根据给定的ID来展现某一个Spittle记录。其中一种方案就是编写处理器方法，通过使用@RequestParam注解，让它接受ID作为查询参数：

```
@RequestMapping(value="/show", method=RequestMethod.GET)
public String showSpittle(
    @RequestParam("spittle_id") long spittleId,
    Model model) {
  model.addAttribute(spittleRepository.findOne(spittleId));
  return "spittle";
}
```

这个处理器方法将会处理形如“/spittles/show?spittle_id=12345”这样的请求。尽管这也可以正常工作，但是从面向资源的角度来看这并不理想。在理想情况下，要识别的资源（Spittle）应该通过URL路径进行标示，而不是通过查询参数。对“/spittles/12345”发起GET请求要优于对“/spittles/show?spittle_id=12345”发起请求。前者能够识别出要查询的资源，而后者描述的是带有参数的一个操作——本质上是通过HTTP发起的RPC。

既然已经以面向资源的控制器作为目标，那我们将这个需求转换为一个测试。程序清单5.12展现了一个新的测试方法，它会断言SpittleController中对面向资源 请求的处理。

程序清单**5.12** 测试对某个**Spittle**的请求，其中**ID**要在路径变量中指定

```
@Test
public void testSpittle() throws Exception {
  Spittle expectedSpittle = new Spittle("Hello", new Date());
  SpittleRepository mockRepository = mock(SpittleRepository.class);
  when(mockRepository.findOne(12345)).thenReturn(expectedSpittle);

  SpittleController controller = new SpittleController(mockRepository);
  MockMvc mockMvc = standaloneSetup(controller).build();

  mockMvc.perform(get("/spittles/12345"))        <-- 通过路径请求资源
    .andExpect(view().name("spittle"))
    .andExpect(model().attributeExists("spittle"))

    .andExpect(model().attribute("spittle", expectedSpittle));
}
```

可以看到，这个测试构建了一个mock Repository、一个控制器和MockMvc，这与本章中我们所编写的其他测试很类似。这个测试中最重要的部分是最后几行，它对“/spittles/12345”发起GET请求，然后断言视图的名称是spittle，并且预期的Spittle对象放到了模型之中。因为我们还没有为这种请求实现处理器方法，因此这个请求将会失败。但是，我们可以通过为SpittleController添加新的方法来修正这个失败的测试。

到目前为止，在我们编写的控制器中，所有的方法都映射到了（通过@RequestMapping）静态定义好的路径上。但是，如果想让这个测试通过的话，我们编写的@RequestMapping要包含变量部分，这部分代表了Spittle ID。

为了实现这种路径变量，Spring MVC允许我们在@RequestMapping路径中添加占位符。占位符的名称要用大括号（“{”和“}”）括起来。路径中的其他部分要与所处理的请求完全匹配，但是占位符部分可以是任意的值。

下面的处理器方法使用了占位符，将Spittle ID作为路径的一部分：

```
@RequestMapping(value="/{spittleId}", method=RequestMethod.GET)
public String spittle(
    @PathVariable("spittleId") long spittleId,
    Model model) {
  model.addAttribute(spittleRepository.findOne(spittleId));
  return "spittle";
}
```

例如，它就能够处理针对“/spittles/12345”的请求，也就是程序清单5.12中的路径

我们可以看到，spittle()方法的spittleId参数上添加了@PathVariable("spittleId")注解，这表明在请求路径中，不管占位符部分的值是什么都会传递到处理器方法的spittleId参数

中。如果对“/spittles/54321”发送GET请求，那么将会把“54321”传递进来，作为spittleId的值。

需要注意的是：在样例中spittleId这个词出现了好几次：先是在@RequestMapping的路径中，然后作为@PathVariable属性的值，最后又作为方法的参数名称。因为方法的参数名碰巧与占位符的名称相同，因此我们可以去掉@PathVariable中的value属性：

```
@RequestMapping(value="/{spittleId}", method=RequestMethod.GET)
public String spittle(@PathVariable long spittleId, Model model) {
  model.addAttribute(spittleRepository.findOne(spittleId));
  return "spittle";
}
```

如果@PathVariable中没有value属性的话，它会假设占位符的名称与方法的参数名相同。这能够让代码稍微简洁一些，因为不必重复写占位符的名称了。但需要注意的是，如果你想要重命名参数时，必须要同时修改占位符的名称，使其互相匹配。

spittle()方法会将参数传递到SpittleRepository的findOne()方法中，用来获取某个Spittle对象，然后将Spittle对象添加到模型中。模型的key将会是spittle，这是根据传递到addAttribute()方法中的类型推断得到的。

这样Spittle对象中的数据就可以渲染到视图中了，此时需要引用请求中key为spittle的属性（与模型的key一致）。如下为渲染Spittle的JSP视图片段：

```
<div class="spittleView">
  <div class="spittleMessage"><c:out value="${spittle.message}" /></div>
  <div>
    <span class="spittleTime"><c:out value="${spittle.time}" /></span>
  </div>
</div>
```

这个视图并没有什么特别之处，它的屏幕截图如图5.4所示。

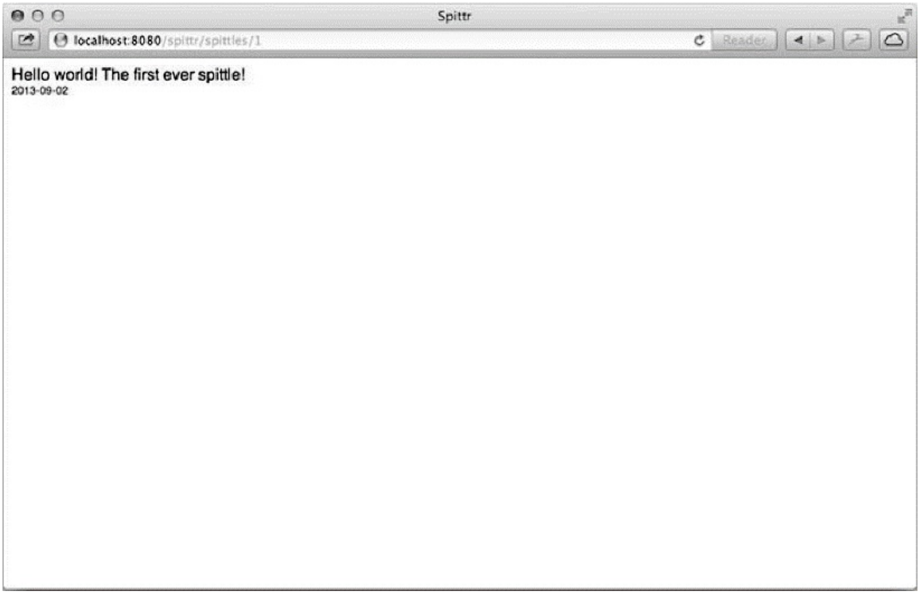

图5.4　在浏览器中展现一个spittle

如果传递请求中少量的数据，那查询参数和路径变量是很合适的。但通常我们还需要传递很多的数据（也许是表单提交的数据），那查询参数显得有些笨拙和受限了。下面让我们来看一下如何编写控制器方法来处理表单提交。

5.4　处理表单

Web应用的功能通常并不局限于为用户推送内容。大多数的应用允许用户填充表单并将数据提交回应用中，通过这种方式实现与用户的交互。像提供内容一样，Spring MVC的控制器也为表单处理提供了良好的支持。

使用表单分为两个方面：展现表单以及处理用户通过表单提交的数据。在Spittr应用中，我们需要有个表单让新用户进行注册。`SpitterController`是一个新的控制器，目前只有一个请求处理的方法来展现注册表单。

程序清单**5.13**　**SpitterController**：展现一个表单，允许用户注册该应用

```
package spittr.web;
import static org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import spittr.Spitter;
import spittr.data.SpitterRepository;

@Controller
@RequestMapping("/spitter")
public class SpitterController {
  @RequestMapping(value="/register", method=GET)      <-- 处理对“/spitter/register”
  public String showRegistrationForm() {                   的 GET 请求
    return "registerForm";
  }

}
```

showRegistrationForm()方法的@RequestMapping注解以及类级别上的@RequestMapping注解组合起来，声明了这个方法要处理的是针对“/spitter/register”的GET请求。这是一个简单的方法，没有任何输入并且只是返回名为registerForm的逻辑视图。按照我们配置InternalResourceViewResolver的方式，这意味着将会使用“/WEB-INF/ views/registerForm.jsp”这个JSP来渲染注册表单。

尽管showRegistrationForm()方法非常简单，但测试依然需要覆盖到它。因为这个方法很简单，所以它的测试也比较简单。

程序清单**5.14**　测试展现表单的控制器方法

```
@Test
public void shouldShowRegistration() throws Exception {
  SpitterController controller = new SpitterController();
  MockMvc mockMvc = standaloneSetup(controller).build();     <-- 构建 MockMvc

  mockMvc.perform(get("/spitter/register"))
         .andExpect(view().name("registerForm"));    <-- 断言 registerForm 视图
}
```

这个测试方法与首页控制器的测试非常类似。它对“/spitter/register”发送GET请求，然后断言结果的视图名为registerForm。

现在，让我们回到视图上。因为视图的名称为registerForm，所以JSP的名称需要是registerForm.jsp。这个JSP必须要包含一个HTML <form>标签，在这个标签中用户输入注册应用的信息。如下就是我们现在所要使用的JSP。

程序清单**5.15**　渲染注册表单的**JSP**

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
  <head>
    <title>Spittr</title>
    <link rel="stylesheet" type="text/css"
          href="<c:url value="/resources/style.css" />" >
  </head>
  <body>
    <h1>Register</h1>

    <form method="POST">
      First Name: <input type="text" name="firstName" /><br/>
      Last Name: <input type="text" name="lastName" /><br/>
      Username: <input type="text" name="username" /><br/>
      Password: <input type="password" name="password" /><br/>

      <input type="submit" value="Register" />
    </form>
  </body>
</html>
```

可以看到，这个JSP非常基础。它的HTML表单域中记录用户的名字、姓氏、用户名以及密码，然后还包含一个提交表单的按钮。在浏览器渲染之后，它的样子大致如图5.5所示。

需要注意的是：这里的`<form>`标签中并没有设置`action`属性。在这种情况下，当表单提交时，它会提交到与展现时相同的URL路径上。也就是说，它会提交到“/spitter/register”上。

这就意味着需要在服务器端处理该HTTP `POST`请求。现在，我们在`Spitter-Controller`中再添加一个方法来处理这个表单提交。

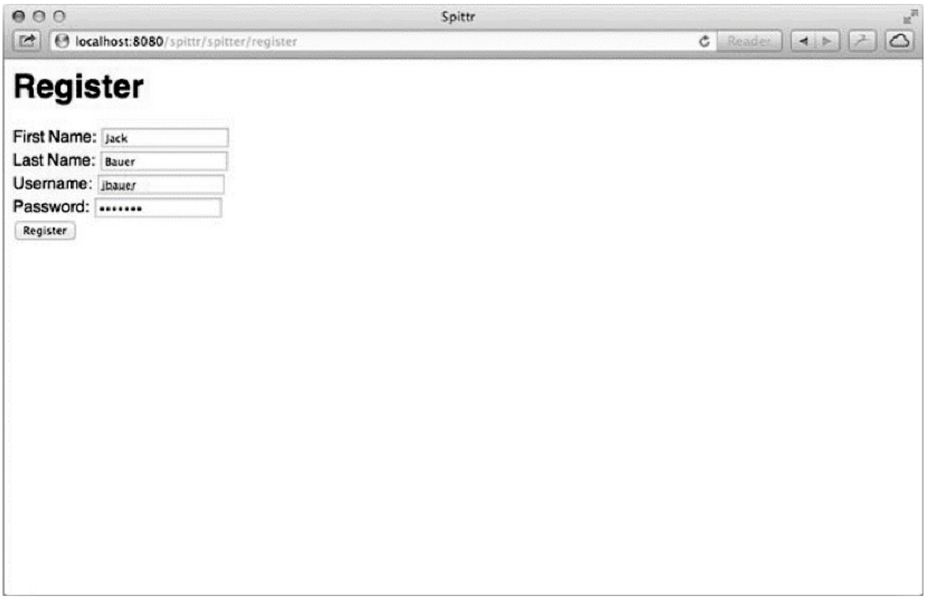

图5.5　注册页提供了一个表单，这个表单会由SpitterController进行处理，完成为应用添加新用户的功能

5.4.1　编写处理表单的控制器

当处理注册表单的POST请求时，控制器需要接受表单数据并将表单数据保存为Spitter对象。最后，为了防止重复提交（用户点击浏览器的刷新按钮有可能会发生这种情况），应该将浏览器重定向到新创建用户的基本信息页面。这些行为通过下面的shouldProcessRegistration()进行了测试。

程序清单5.16　测试处理表单的控制器方法

```
@Test
public void shouldProcessRegistration() throws Exception {
  SpitterRepository mockRepository =
          mock(SpitterRepository.class);      <-- 构建 Repository
  Spitter unsaved =
          new Spitter("jbauer", "24hours", "Jack", "Bauer");
  Spitter saved =
          new Spitter(24L, "jbauer", "24hours", "Jack", "Bauer");
  when(mockRepository.save(unsaved)).thenReturn(saved);

  SpitterController controller =
          new SpitterController(mockRepository);
  MockMvc mockMvc = standaloneSetup(controller).build();   <-- 构建 MockMvc

  mockMvc.perform(post("/spitter/register")      <-- 执行请求
         .param("firstName", "Jack")
         .param("lastName", "Bauer")
         .param("username", "jbauer")
         .param("password", "24hours"))
         .andExpect(redirectedUrl("/spitter/jbauer"));

  verify(mockRepository, atLeastOnce()).save(unsaved);      <-- 校验保存情况
}
```

显然，这个测试比展现注册表单的测试复杂得多。在构建完SpitterRepository的mock实现以及所要执行的控制器和MockMvc之后，shouldProcess-Registration()对"/spitter/register"发起了一个POST请求。作为请求的一部分，用户信息以参数的形式放到request中，从而模拟提交的表单。

在处理POST类型的请求时，在请求处理完成后，最好进行一下重定向，这样浏览器的刷新就不会重复提交表单了。在这个测试中，预期请求会重定向到"/spitter/jbauer"，也就是新建用户的基本信息页面。

最后，测试会校验SpitterRepository的mock实现最终会真正用来保存表单上传入的数据。

现在，我们来实现处理表单提交的控制器方法。通过shouldProcess-Registration()方法，我们可能认为要满足这个需求需要做很多的工作。但是，在如下的程序清单中，我们可以看到新的SpitterController并没有做太多的事情。

程序清单**5.17**　处理所提交的表单并注册新用户

```
package spittr.web;

import static org.springframework.web.bind.annotation.RequestMethod.*;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import spittr.Spitter;
import spittr.data.SpitterRepository;

@Controller
@RequestMapping("/spitter")
public class SpitterController {
  private SpitterRepository spitterRepository;

  @Autowired
  public SpitterController(                        <-- 注入 SpitterRepository
      SpitterRepository spitterRepository) {
    this.spitterRepository = spitterRepository;
  }

  @RequestMapping(value="/register", method=GET)
  public String showRegistrationForm() {
    return "registerForm";
  }

  @RequestMapping(value="/register", method=POST)
  public String processRegistration(Spitter spitter) {
    spitterRepository.save(spitter);                 <-- 保存 Spitter

    return "redirect:/spitter/" +                 <-- 重定向到基本信息页
           spitter.getUsername();
  }
}
```

我们之前创建的showRegistrationForm()方法依然还在，不过请注意新创建的processRegistration()方法，它接受一个Spitter对象作为参数。这个对象有firstName、lastName、username和password属性，这些属性将会使用请求中同名的参数进行填充。

当使用Spitter对象调用processRegistration()方法时，它会进而调用SpitterRepository的save()方法，SpitterRepository是在Spitter-Controller的构造器中注入进来的。

processRegistration()方法做的最后一件事就是返回一个String类型，用来指定视图。但是这个视图格式和以前我们所看到的视图有所不同。这里不仅返回了视图的名称供视图解析器查找目标视图，而且返回的值还带有重定向的格式。

当InternalResourceViewResolver看到视图格式中的“redirect:”前缀时，它就知道要将其解析为重定向的规则，而不是

视图的名称。在本例中，它将会重定向到用户基本信息的页面。例如，如果Spitter.username属性的值为“jbauer”，那么视图将会重定向到“/spitter/jbauer”。

需要注意的是，除了“redirect:”，InternalResourceViewResolver还能识别“forward:”前缀。当它发现视图格式中以“forward:”作为前缀时，请求将会前往（forward）指定的URL路径，而不再是重定向。

万事俱备！现在，程序清单5.16中的测试应该能够通过了。但是，我们的任务还没有完成，因为我们重定向到了用户基本信息页面，那么我们应该往SpitterController中添加一个处理器方法，用来处理对基本信息页面的请求。如下的showSpitterProfile()将会完成这项任务：

```
@RequestMapping(value="/{username}", method=GET)

public String showSpitterProfile(
      @PathVariable String username, Model model) {
  Spitter spitter = spitterRepository.findByUsername(username);
  model.addAttribute(spitter);
  return "profile";
}
```

SpitterRepository通过用户名获取一个Spitter对象，showSpitter-Profile()得到这个对象并将其添加到模型中，然后返回profile，也就是基本信息页面的逻辑视图名。像本章展现的其他视图一样，现在的基本信息视图非常简单：

```
<h1>Your Profile</h1>
<c:out value="${spitter.username}" /><br/>
<c:out value="${spitter.firstName}" />
    <c:out value="${spitter.lastName}" />
```

图5.6展现了在Web浏览器中渲染的基本信息页面。

如果表单中没有发送username或password的话，会发生什么情况呢？或者说，如果firstName或lastName的值为空或太长的话，又会怎么样呢？接下来，让我们看一下如何为表单提交添加校验，从而避免数据呈现的不一致性。

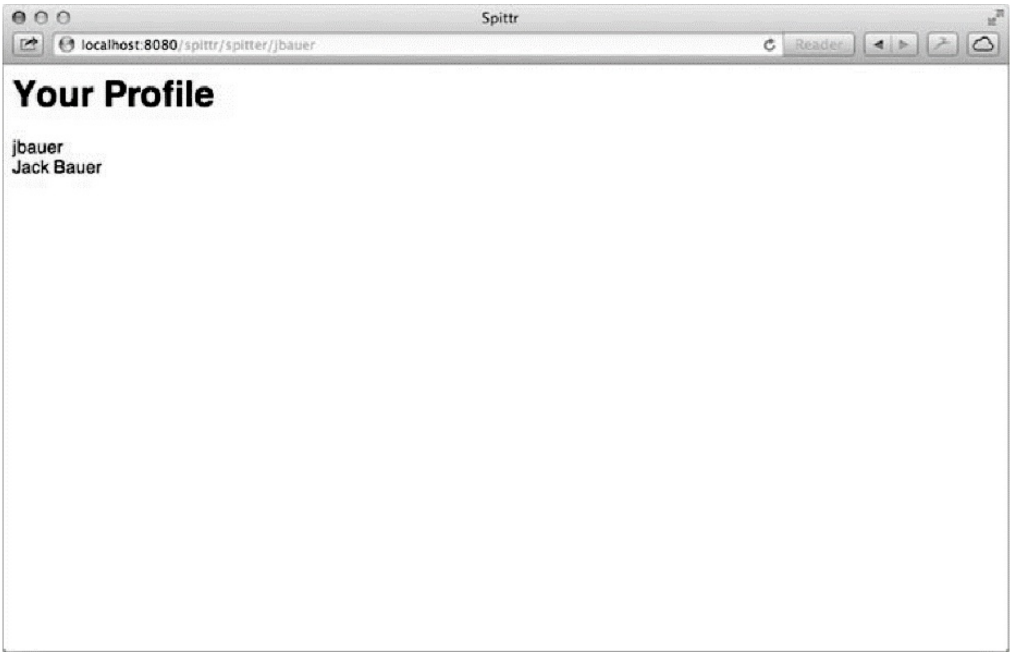

图5.6　Spittr的基本信息页展现了用户的情况，这些信息是由SpitterController填充到模型中的

5.4.2　校验表单

如果用户在提交表单的时候，username或password文本域为空的话，那么将会导致在新建Spitter对象中，username或password是空的String。至少这是一种怪异的行为。如果这种现象不处理的话，这将会出现安全问题，因为不管是谁只要提交一个空的表单就能登录应用。

同时，我们还应该阻止用户提交空的firstName和/或lastName，使应用仅在一定程度上保持匿名性。有个好的办法就是限制这些输入域值的长度，保持它们的值在一个合理的长度范围，避免这些输入域的误用。

有种处理校验的方式非常初级，那就是在processRegistration()方法中添加代码来检查值的合法性，如果值不合法的话，就将注册表单重新显示给用户。这是一个很简短的方法，因此，添加一些额外的if语句也不是什么大问题，对吧？

与其让校验逻辑弄乱我们的处理器方法，还不如使用Spring对Java校验API（Java Validation API，又称JSR-303）的支持。从Spring 3.0开始，在Spring MVC中提供了对Java校验API的支持。在Spring MVC中要使用Java校验API的话，并不需要什么额外的配置。只要保证在类路径下包含这个Java API的实现即可，比如Hibernate Validator。

Java校验API定义了多个注解，这些注解可以放到属性上，从而限制这些属性的值。所有的注解都位于javax.validation.constraints包中。表5.1列出了这些校验注解。

表**5.1　Java**校验**API**所提供的校验注解

注　　解	描　　述
@AssertFalse	所注解的元素必须是Boolean类型，并且值为false
@AssertTrue	所注解的元素必须是Boolean类型，并且值为true
@DecimalMax	所注解的元素必须是数字，并且它的值要小于或等于给定的BigDecimalString值
@DecimalMin	所注解的元素必须是数字，并且它的值要大于或等于给定的BigDecimalString值
@Digits	所注解的元素必须是数字，并且它的值必须有指定的位数
@Future	所注解的元素的值必须是一个将来的日期
@Max	所注解的元素必须是数字，并且它的值要小于或等于给定的值
@Min	所注解的元素必须是数字，并且它的值要大于或等于给定的值
@NotNull	所注解元素的值必须不能为null
@Null	所注解元素的值必须为null
@Past	所注解的元素的值必须是一个已过去的日期
@Pattern	所注解的元素的值必须匹配给定的正则表达式

@Size	所注解的元素的值必须是String、集合或数组，并且它的长度要符合给定的范围

除了表5.1中的注解，Java校验API的实现可能还会提供额外的校验注解。同时，也可以定义自己的限制条件。但就我们来讲，将会关注于上表中的两个核心限制条件。

请考虑要添加到Spitter域上的限制条件，似乎需要使用@NotNull和@Size注解。我们所要做的事情就是将这些注解添加到Spitter的属性上。如下的程序清单展现了Spitter类，它的属性已经添加了校验注解。

程序清单**5.18　Spitter**：包含了要提交到**Spittle POST**请求中的域

```
package spittr;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import org.apache.commons.lang3.builder.EqualsBuilder;
import org.apache.commons.lang3.builder.HashCodeBuilder;

public class Spitter {

  private Long id;

  @NotNull
  @Size(min=5, max=16)                    非空，5 到 16 个字符
  private String username;

  @NotNull
  @Size(min=5, max=25)                    非空，5 到 25 个字符
  private String password;

  @NotNull
  @Size(min=2, max=30)                    非空，2 到 30 个字符
  private String firstName;

  @NotNull
  @Size(min=2, max=30)                    非空，2 到 30 个字符
  private String lastName;

  ...
}
```

现在，Spitter的所有属性都添加了@NotNull注解，以确保它们的值不为null。类似地，属性上也添加了@Size注解以限制它们的长度在最大值和最小值之间。对Spittr应用来说，这意味着用户必须要填完注册表单，并且值的长度要在给定的范围内。

我们已经为Spitter添加了校验注解，接下来需要修改processRegistration()方法来应用校验功能。启用校验功能的processRegistration()如下所示：

程序清单**5.19　processRegistration()**：确保所提交的数据是合法的

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    @Valid Spitter spitter,                    <-- 校验 Spitter 输入
     Errors errors) {

  if (errors.hasErrors()) {                    如果校验出现错误，
    return "registerForm";                  <-- 则重新返回表单
   }

  spitterRepository.save(spitter);
  return "redirect:/spitter/" + spitter.getUsername();
}
```

与程序清单5.17中最初的`processRegistration()`方法相比，这里有了很大的变化。`Spitter`参数添加了`@Valid`注解，这会告知`Spring`，需要确保这个对象满足校验限制。

在`Spitter`属性上添加校验限制并不能阻止表单提交。即便用户没有填写某个域或者某个域所给定的值超出了最大长度，`processRegistration()`方法依然会被调用。这样，我们就需要处理校验的错误，就像在`processRegistration()`方法中所看到的那样。

如果有校验出现错误的话，那么这些错误可以通过`Errors`对象进行访问，现在这个对象已作为`processRegistration()`方法的参数。（很重要一点需要注意，`Errors`参数要紧跟在带有`@Valid`注解的参数后面，`@Valid`注解所标注的就是要检验的参数。）`processRegistration()`方法所做的第一件事就是调用`Errors.hasErrors()`来检查是否有错误。

如果有错误的话，`Errors.hasErrors()`将会返回到`registerForm`，也就是注册表单的视图。这能够让用户的浏览器重新回到注册表单页面，所以他们能够修正错误，然后重新尝试提交。现在，会显示空的表单，但是在下一章中，我们将在表单中显示最初提交的值并将校验错误反馈给用户。

如果没有错误的话，`Spitter`对象将会通过Repository进行保存，控制器会像之前那样重定向到基本信息页面。

5.5　小结

在本章中，我们为编写应用程序的Web部分开了一个好头。可以看到，Spring有一个强大灵活的Web框架。借助于注解，Spring MVC提

供了近似于POJO的开发模式，这使得开发处理请求的控制器变得非常简单，同时也易于测试。

当编写控制器的处理器方法时，Spring MVC极其灵活。概括来讲，如果你的处理器方法需要内容的话，只需将对应的对象作为参数，而它不需要的内容，则没有必要出现在参数列表中。这样，就为请求处理带来了无限的可能性，同时还能保持一种简单的编程模型。

尽管本章中的很多内容都是关于控制器的请求处理的，但是渲染响应同样也是很重要的。我们通过使用JSP的方式，简单了解了如何为控制器编写视图。但是就Spring MVC的视图来说，它并不限于本章所看到的简单JSP。

在接下来的第6章中，我们将会更深入地学习Spring视图，包括如何在JSP中使用Spring标签库。我们还会学习如何借助Apache Tiles为视图添加一致的布局结构。同时，还会了解Thymeleaf，这是一个很有意思的JSP替代方案，Spring为其提供了内置的支持。

第6章　渲染Web视图

本章内容：

- 将模型数据渲染为HTML
- 使用JSP视图
- 通过tiles定义视图布局
- 使用Thymeleaf视图

上一章主要关注于如何编写处理Web请求的控制器。我们也创建了一些简单的视图，用来渲染控制器产生的模型数据，但我们并没有花太多时间讨论视图，也没有讨论控制器完成请求到结果渲染到用户的浏览器中的这段时间内到底发生了什么，而这正是本章的主要内容。

6.1　理解视图解析

在第5章中，我们所编写的控制器方法都没有直接产生浏览器中渲染所需的HTML。这些方法只是将一些数据填充到模型中，然后将模型传递给一个用来渲染的视图。这些方法会返回一个String类型的值，这个值是视图的逻辑名称，不会直接引用具体的视图实现。尽管我们也编写了几个简单的JavaServer Page（JSP）视图，但是控制器并不关心这些。

将控制器中请求处理的逻辑和视图中的渲染实现解耦是Spring MVC的一个重要特性。如果控制器中的方法直接负责产生HTML的话，就很难在不影响请求处理逻辑的前提下，维护和更新视图。控制器方法和视图的实现会在模型内容上达成一致，这是两者的最大关联，除此之外，两者应该保持足够的距离。

但是，如果控制器只通过逻辑视图名来了解视图的话，那Spring该如何确定使用哪一个视图实现来渲染模型呢？这就是Spring视图解析器的任务了。

在第5章中，我们使用名为`InternalResourceViewResolver`的视图解析器。在它的配置中，为了得到视图的名字，会使用“/WEB-INF/views/”前缀和“.jsp”后缀，从而确定来渲染模型的JSP文件的物理位置。现在，我们回过头来看一下视图解析的基础知识以及Spring提

供的其他视图解析器。

Spring MVC定义了一个名为ViewResolver的接口，它大致如下所示：

```
public interface ViewResolver {
  View resolveViewName(String viewName, Locale locale)
                     throws Exception;
}
```

当给resolveViewName()方法传入一个视图名和Locale对象时，它会返回一个View实例。View是另外一个接口，如下所示：

```
public interface View {
  String getContentType();
  void render(Map<String, ?> model,
              HttpServletRequest request,
              HttpServletResponse response) throws Exception;
}
```

View接口的任务就是接受模型以及Servlet的request和response对象，并将输出结果渲染到response中。

这看起来非常简单。我们所需要做的就是编写ViewResolver和View的实现，将要渲染的内容放到response中，进而展现到用户的浏览器中。对吧？

实际上，我们并不需要这么麻烦。尽管我们可以编写ViewResolver和View的实现，在有些特定的场景下，这样做也是有必要的，但是一般来讲，我们并不需要关心这些接口。我在这里提及这些接口只是为了让你对视图解析内部如何工作有所了解。Spring提供了多个内置的实现，如表6.1所示，它们能够适应大多数的场景。

表**6.1** **Spring**自带了**13**个视图解析器，能够将逻辑视图名转换为物理实现

视图解析器	描　　述
BeanNameViewResolver	将视图解析为Spring应用上下文中的bean，其中bean的ID与视图的名字相同
ContentNegotiatingViewResolver	通过考虑客户端需要的内容类型来解析视图，委托给另外一个能够产生对应内容类型的视图解析器

FreeMarkerViewResolver	将视图解析为FreeMarker模板
InternalResourceViewResolver	将视图解析为Web应用的内部资源（一般为JSP）
JasperReportsViewResolver	将视图解析为JasperReports定义
ResourceBundleViewResolver	将视图解析为资源bundle（一般为属性文件）
TilesViewResolver	将视图解析为Apache Tile定义，其中tile ID与视图名称相同。注意有两个不同的TilesViewResolver实现，分别对应于Tiles 2.0和Tiles 3.0
UrlBasedViewResolver	直接根据视图的名称解析视图，视图的名称会匹配一个物理视图的定义
VelocityLayoutViewResolver	将视图解析为Velocity布局，从不同的Velocity模板中组合页面
VelocityViewResolver	将视图解析为Velocity模板
XmlViewResolver	将视图解析为特定XML文件中的bean定义。类似于BeanName-ViewResolver
XsltViewResolver	将视图解析为XSLT转换后的结果

Spring 4和Spring 3.2支持表6.1中的所有视图解析器。Spring 3.1支持除Tiles 3 TilesViewResolver之外的所有视图解析器。

我们没有足够的篇幅介绍Spring所提供的13种视图解析器。这其实也没什么，因为在大多数应用中，我们只会用到其中很少的一部分。

对于表6.1中的大部分视图解析器来讲，每一项都对应Java Web应用中特定的某种视图技术。InternalResourceViewResolver一般会用于JSP，TilesViewResolver用于Apache Tiles视图，

而FreeMarkerViewResolver和VelocityViewResolver分别对应FreeMarker和Velocity模板视图。

在本章中，我们将会关注与大多数Java开发人员最息息相关的视图技术。因为大多数Java Web应用都会用到JSP，我们首先将会介绍InternalResourceViewResolver，这个视图解析器一般会用来解析JSP视图。接下来，我们将会介绍TilesViewResolver，控制JSP页面的布局。

在本章的最后，我们将会看一个没有列在表6.1中的视图解析器。Thymeleaf是一种用来替代JSP的新兴技术，Spring提供了与Thymeleaf的原生模板（natural template）协作的视图解析器，这种模板之所以得到这样的称呼是因为它更像是最终产生的HTML，而不是驱动它们的Java代码。Thymeleaf是一种非常令人兴奋的视图方案，所以你尽可以先往后翻几页，去6.4节看一下在Spring中是如何使用 它的。

如果你依然停留在本页的话，那么你可能知道JSP曾经是，而且现在依然还是Java领域占主导地位的视图技术。在以前的项目中，也许你使用过JSP，将来有可能还会继续使用这项技术，所以接下来让我们看一下如何在Spring MVC中使用JSP 视图。

6.2 创建JSP视图

不管你是否相信，JavaServer Pages作为Java Web应用程序的视图技术已经超过15年了。尽管开始的时候它很丑陋，只是类似模板技术（如Microsoft的Active Server Pages）的Java版本，但JSP这些年在不断进化，包含了对表达式语言和自定义标签库的支持。

Spring提供了两种支持JSP视图的方式：

- InternalResourceViewResolver会将视图名解析为JSP文件。另外，如果在你的JSP页面中使用了JSP标准标签库（JavaServer Pages Standard Tag Library，JSTL）的话，InternalResourceViewResolver能够将视图名解析为JstlView形式的JSP文件，从而将JSTL本地化和资源bundle变量暴露给JSTL的格式化（formatting）和信息（message）标签。
- Spring提供了两个JSP标签库，一个用于表单到模型的绑定，另一个提供了通用的工具类特性。

不管你使用JSTL，还是准备使用Spring的JSP标签库，配置解析JSP的

视图解析器都是非常重要的。尽管Spring还有其他的几个视图解析器都能将视图名映射为JSP文件，但就这项任务来讲，InternalResourceViewResolver是最简单和最常用的视图解析器。我们在第5章已经接触到了如何配置InternalResourceViewResolver。但是在那里，我们只是匆忙体验了一下，以便于查看控制器在浏览器中的效果。接下来，我们将会更加仔细地了解InternalResourceViewResolver，看看如何让它完全听命于我们。

6.2.1 配置适用于JSP的视图解析器

有一些视图解析器，如ResourceBundleViewResolver会直接将逻辑视图名映射为特定的View接口实现，而InternalResourceViewResolver所采取的方式并不那么直接。它遵循一种约定，会在视图名上添加前缀和后缀，进而确定一个Web应用中视图资源的物理路径。

作为样例，考虑一个简单的场景，假设逻辑视图名为home。通用的实践是将JSP文件放到Web应用的WEB-INF目录下，防止对它的直接访问。如果我们将所有的JSP文件都放在“/WEB-INF/views/”目录下，并且home页的JSP名为home.jsp，那么我们可以确定物理视图的路径就是逻辑视图名home再加上“/WEB-INF/views/”前缀和“.jsp”后缀。如图6.1所示。

图6.1 nternalResourceViewResolver解析视图时，会在视图名上添加前缀和后缀

当使用@Bean注解的时候，我们可以按照如下的方式配置Internal-ResourceView Resolver，使其在解析视图时，遵循上述的约定。

```
@Bean
public ViewResolver viewResolver() {
  InternalResourceViewResolver resolver =
      new InternalResourceViewResolver();
  resolver.setPrefix("/WEB-INF/views/");
  resolver.setSuffix(".jsp");
  return resolver;
}
```

作为替代方案，如果你更喜欢使用基于XML的Spring配置，那么可以按照如下的方式配置`InternalResourceViewResolver`：

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
                      InternalResourceViewResolver"
      p:prefix="/WEB-INF/views/"
      p:suffix=".jsp" />
```

`InternalResourceViewResolver`配置就绪之后，它就会将逻辑视图名解析为JSP文件，如下所示：

- `home`将会解析为“/WEB-INF/views/home.jsp”
- `productList`将会解析为“/WEB-INF/views/productList.jsp”
- `books/detail`将会解析为“/WEB-INF/views/books/detail.jsp”

让我们重点看一下最后一个样例。当逻辑视图名中包含斜线时，这个斜线也会带到资源的路径名中。因此，它会对应到`prefix`属性所引用目录的子目录下的JSP文件。这样的话，我们就可以很方便地将视图模板组织为层级目录结构，而不是将它们都放到同一个目录之中。

解析**JSTL**视图

到目前为止，我们对`InternalResourceViewResolver`的配置都很基础和简单。它最终会将逻辑视图名解析为`InternalResourceView`实例，这个实例会引用JSP文件。但是如果这些JSP使用JSTL标签来处理格式化和信息的话，那么我们会希望`InternalResourceViewResolver`将视图解析为`JstlView`。

JSTL的格式化标签需要一个`Locale`对象，以便于恰当地格式化地域相关的值，如日期和货币。信息标签可以借助Spring的信息资源和`Locale`，从而选择适当的信息渲染到HTML之中。通过解析`JstlView`，JSTL能够获得`Locale`对象以及Spring中配置的信息资源。

如果想让`InternalResourceViewResolver`将视图解析为`JstlView`，而不是`InternalResourceView`的话，那么我们只

需设置它的viewClass属性即可：

```
@Bean
public ViewResolver viewResolver() {
  InternalResourceViewResolver resolver =
      new InternalResourceViewResolver();
  resolver.setPrefix("/WEB-INF/views/");
  resolver.setSuffix(".jsp");
  resolver.setViewClass(
      org.springframework.web.servlet.view.JstlView.class);
  return resolver;
}
```

同样，我们也可以使用XML完成这一任务：

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
                      InternalResourceViewResolver"
      p:prefix="/WEB-INF/views/"
      p:suffix=".jsp"
      p:viewClass="org.springframework.web.servlet.view.JstlView" />
```

不管使用Java配置还是使用XML，都能确保JSTL的格式化和信息标签能够获得Locale对象以及Spring中配置的信息资源。

6.2.2 使用Spring的JSP库

当为JSP添加功能时，标签库是一种很强大的方式，能够避免在脚本块中直接编写Java代码。Spring提供了两个JSP标签库，用来帮助定义Spring MVC Web的视图。其中一个标签库会用来渲染HTML表单标签，这些标签可以绑定model中的某个属性。另外一个标签库包含了一些工具类标签，我们随时都可以非常便利地使用它们。

在这两个标签库中，你可能会发现表单绑定的标签库更加有用。所以，我们就从这个标签库开始学习Spring的JSP标签。我们将会看到如何将Spittr应用的注册表单绑定到模型上，这样表单就可以预先填充值，并且在表单提交失败后，能够展现校验错误。

将表单绑定到模型上

Spring的表单绑定JSP标签库包含了14个标签，它们中的大多数都用来渲染HTML中的表单标签。但是，它们与原生HTML标签的区别在于它们会绑定模型中的一个对象，能够根据模型中对象的属性填充值。标签库中还包含了一个为用户展现错误的标签，它会将错误信息渲染到最终的HTML之中。

为了使用表单绑定库，需要在JSP页面中对其进行声明：

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="sf" %>
```

需要注意，我将前缀指定为“sf”，但通常也可能使用“form”前缀。你可以选择任意喜欢的前缀，我之所以选择“sf”是因为它很简洁、易于输入，并且还是Spring form的简写形式。在本书中，当使用表单绑定库的时候，我会一直使用“sf”前缀。

在声明完表单绑定标签库之后，你就可以使用14个相关的标签了。如表6.2所示。

表**6.2** 借助**Spring**表单绑定标签库中所包含的标签，我们能够将模型对象绑定到渲染后的**HTML**表单中

JSP标签	描　　述
<sf:checkbox>	渲染成一个HTML <input>标签，其中type属性设置为checkbox
<sf:checkboxes>	渲染成多个HTML <input>标签，其中type属性设置为checkbox
<sf:errors>	在一个HTML <span>中渲染输入域的错误
<sf:form>	渲染成一个HTML <form>标签，并为其内部标签暴露绑定路径，用于数据绑定
<sf:hidden>	渲染成一个HTML <input>标签，其中type属性设置为hidden
<sf:input>	渲染成一个HTML <input>标签，其中type属性设置为text
<sf:label>	渲染成一个HTML <label>标签
<sf:option>	渲染成一个HTML <option>标签，其selected属性根据所绑定的值进行设置
<sf:options>	按照绑定的集合、数组或Map，渲染成一个HTML <option>标签的列表

<sf:password>	渲染成一个HTML <input>标签，其中type属性设置为password
<sf:radiobutton>	渲染成一个HTML <input>标签，其中type属性设置为radio
<sf:radiobuttons>	渲染成多个HTML <input>标签，其中type属性设置为radio
<sf:select>	渲染为一个HTML <select>标签
<sf:textarea>	渲染为一个HTML <textarea>标签

要在一个样例中介绍所有的这些标签是很困难的，如果一定要这样做的话，肯定也会非常牵强。就Spittr样例来说，我们只会用到适合于Spittr应用中注册表单的标签。具体来讲，也就是<sf:form>、<sf:input>和<sf:password>。在注册JSP中使用这些标签后，所得到的程序如下所示：

```
<sf:form method="POST" commandName="spitter">
  First Name: <sf:input path="firstName" /><br/>
  Last Name: <sf:input path="lastName" /><br/>
  Email: <sf:input path="email" /><br/>
  Username: <sf:input path="username" /><br/>
  Password: <sf:password path="password" /><br/>
  <input type="submit" value="Register" />
</sf:form>
```

<sf:form>会渲染会一个HTML <form>标签，但它也会通过commandName属性构建针对某个模型对象的上下文信息。在其他的表单绑定标签中，会引用这个模型对象的属性。

在之前的代码中，我们将commandName属性设置为spitter。因此，在模型中必须要有一个key为spitter的对象，否则的话，表单不能正常渲染（会出现JSP错误）。这意味着我们需要修改一下SpitterController，以确保模型中存在以spitter为key的Spitter对象：

```
@RequestMapping(value="/register", method=GET)
public String showRegistrationForm(Model model) {
  model.addAttribute(new Spitter());
  return "registerForm";
}
```

修改后的showRegistrationForm()方法中，新增了一个Spitter实例到模型中。模型中的key是根据对象类型推断得到的，也就是spitter，与我们所需要的完全一致。

回到这个表单中，前四个输入域将HTML <input>标签改成了<sf:input>。这个标签会渲染成一个HTML <input>标签，并且type属性将会设置为text。我们在这里设置了path属性，<input>标签的value属性值将会设置为模型对象中path属性所对应的值。例如，如果在模型中Spitter对象的firstName属性值为Jack，那么<sf:input path="firstName"/>所渲染的<input>标签中，会存在value="Jack"。

对于password输入域，我们使用<sf:password>来代替<sf:input>。<sf:password>与<sf:input>类似，但是它所渲染的HTML <input>标签中，会将type属性设置为password，这样当输入的时候，它的值不会直接明文显示。

为了帮助读者了解最终的HTML看起来是什么样子的，假设有个用户已经提交了表单，但值都是不合法的。校验失败后，用户会被重定向到注册表单，最终的HTML<form>元素如下所示：

```
<form id="spitter" action="/spitter/spitter/register" method="POST">
  First Name:
      <input id="firstName"
             name="firstName" type="text" value="J"/><br/>
  Last Name:
      <input id="lastName"
             name="lastName" type="text" value="B"/><br/>
  Email:
      <input id="email"
             name="email" type="text" value="jack"/><br/>
  Username:
      <input id="username"
             name="username" type="text" value="jack"/><br/>
  Password:
      <input id="password"
             name="password" type="password" value=""/><br/>
  <input type="submit" value="Register" />
</form>
```

值得注意的是，从Spring 3.1开始，<sf:input>标签能够允许我们指定type属性，这样的话，除了其他可选的类型外，还能指定HTML 5特定类型的文本域，如date、range和email。例如，我们可以按照如下的方式指定email域：

```
Email: <sf:input path="email" type="email" /><br/>
```

这样所渲染得到的HTML如下所示：

```
Email:
    <input id="email" name="email" type="email" value="jack"/><br/>
```

相对于标准的HTML标签，使用Spring的表单绑定标签能够带来一定的功能提升，在校验失败后，表单中会预先填充之前输入的值。但是，这依然没有告诉用户错在什么地方。为了指导用户矫正错误，我们需要使用`<sf:errors>`。

展现错误

如果存在校验错误的话，请求中会包含错误的详细信息，这些信息是与模型数据放到一起的。我们所需要做的就是到模型中将这些数据抽取出来，并展现给用户。`<sf:errors>`能够让这项任务变得很简单。

例如，让我们看一下将`<sf:errors>`用到registerForm.jsp中的代码片段：

```
<sf:form method="POST" commandName="spitter">
  First Name: <sf:input path="firstName" />
    <sf:errors path="firstName" /><br/>
...
</sf:form>
```

尽管我只展现了将`<sf:errors>`用到First Name输入域的场景，但是它可以按照同样简单的方式用到注册表单的其他输入域中。在这里，它的`path`属性设置成了`firstName`，也就是指定了要显示`Spitter`模型对象中哪个属性的错误。如果`firstName`属性没有错误的话，那么`<sf:errors>`不会渲染任何内容。但如果有校验错误的话，那么它将会在一个HTML `<span>`标签中显示错误信息。

例如，如果用户提交字母“J”作为名字的话，那么如下的HTML片段就是针对First Name输入域所显示的内容：

```
First Name: <input id="firstName"
                   name="firstName" type="text" value="J"/>
<span id="firstName.errors">size must be between 2 and 30</span>
```

现在，我们已经可以为用户展现错误信息，这样他们就能修正这些错误了。我们可以更进一步，修改错误的样式，使其更加突出显示。为了做到这一点，可以设置cssClass属性：

```
<sf:form method="POST" commandName="spitter" >
  First Name: <sf:input path="firstName" />
    <sf:errors path="firstName" cssClass="error" /><br/>
...
</sf:form>
```

同样，简单起见，我只会展现如何为firstName输入域的<sf:errors>设置cssClass属性。你可以将其用到其他的输入域上。

现在errors的<span>会有一个值为error的class属性。剩下需要做的就是为这个类定义CSS样式。如下就是一个简单的CSS样式，它会将错误信息渲染为红色：

```
span.error {
  color: red;
}
```

图6.2展现了这个表单此时在浏览器中的显式效果。

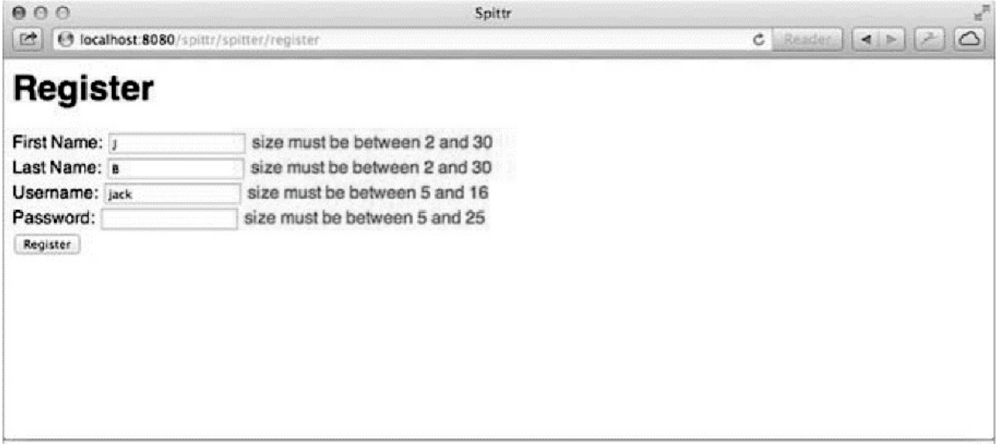

图6.2　在表单输入域的旁边展现校验错误信息

在输入域的旁边展现错误信息是一种很好的方式，这样能够引起用户的关注，提醒他们修正错误。但这样也会带来布局的问题。另外一种处理校验错误方式就是将所有的错误信息在同一个地方进行显示。为了做到这一点，我们可以移除每个输入域上的<sf:errors>元素，并将其放到表单的顶部，如下所示：

```
<sf:form method="POST" commandName="spitter" >
  <sf:errors path="*" element="div" cssClass="errors" />

...
</sf:form>
```

这个<sf:errors>与之前相比，值得注意的不同之处在于它的path

被设置成了“*”。这是一个通配符选择器，会告诉<sf:errors>展现所有属性的所有错误。

同样需要注意的是，我们将element属性设置成了div。默认情况下，错误都会渲染在一个HTML <span>标签中，如果只显示一个错误的话，这是不错的选择。但是，如果要渲染所有输入域的错误的话，很可能要展现不止一个错误，这时候使用<span>标签（行内元素）就不合适了。像<div>这样的块级元素会更为合适。因此，我们可以将element属性设置为div，这样的话，错误就会渲染在一个<div>标签中。

像之前一样，cssClass属性被设置errors，这样我们就能为<div>设置样式。如下为<div>的CSS样式，它具有红色的边框和浅红色的背景：

```
div.errors {
  background-color: #ffcccc;
  border: 2px solid red;
}
```

现在，我们在表单的上方显示所有的错误，这样页面布局可能会更加容易一些。但是，我们还没有着重显示需要修正的输入域。通过为每个输入域设置cssErrorClass属性，这个问题很容易解决。我们也可以将每个label都替换为<sf: label>，并设置它的cssErrorClass属性。如下就是做完必要修改后的First Name输入域：

```
<sf:form method="POST" commandName="spitter" >
  <sf:label path="firstName"
      cssErrorClass="error">First Name</sf:label>:
    <sf:input path="firstName" cssErrorClass="error" /><br/>
...
</sf:form>
```

<sf: label>标签像其他的表单绑定标签一样，使用path来指定它属于模型对象中的哪个属性。在本例中，我们将其设置为firstName，因此它会绑定Spitter对象的firstName属性。假设没有校验错误的话，它将会渲染为如下的HTML<label>元素：

```
<label for="firstName">First Name</label>
```

就其自身来说，设置<sf:label>的path属性并没有完成太多的功能。但是，我们还同时设置了cssErrorClass属性。如果它所绑定的属性有任何错误的话，在渲染得到的<label>元素中，class属性将会被设置为error，如下所示：

```
<label for="firstName" class="error">First Name</label>
```

与之类似，<sf:input>标签的cssErrorClass属性被设置为error。如果有任何校验错误的话，在渲染得到的<input>标签中，class属性将会被设置为error。现在我们已经为文本标记和输入域设置了样式，这样当出现错误的时候，会将用户的注意力转移到此处。例如，如下的CSS会将文本标记渲染为红色，并将输入域设置为浅红色背景：

```
label.error {
  color: red;
}

input.error {
  background-color: #ffcccc;
}
```

现在，我们有了很好的方式为用户展现错误信息。不过，我们还可以做另外一件事情，能够让这些错误信息更加易读。重新看一下Spitter类，我们可以在校验注解上设置message属性，使其引用对用户更为友好的信息，而这些信息可以定义在属性文件中：

```
@NotNull
@Size(min=5, max=16, message="{username.size}")
private String username;

@NotNull
@Size(min=5, max=25, message="{password.size}")
private String password;

@NotNull
@Size(min=2, max=30, message="{firstName.size}")
private String firstName;

@NotNull
@Size(min=2, max=30, message="{lastName.size}")
private String lastName;

@NotNull
@Email(message="{email.valid}")
private String email;
```

对于上面每个域，我们都将其@Size注解的message设置为一个字符串，这个字符串是用大括号括起来的。如果没有大括号的话，message中的值将会作为展现给用户的错误信息。但是使用了大括号之后，我们使用的就是属性文件中的某一个属性，该属性包含了实际的信息。

接下来需要做的就是创建一个名为ValidationMessages.properties的文件，并将其放在根类路径之下：

```
firstName.size=
    First name must be between {min} and {max} characters long.
lastName.size=
    Last name must be between {min} and {max} characters long.
username.size=
    Username must be between {min} and {max} characters long.
password.size=
    Password must be between {min} and {max} characters long.
email.valid=The email address must be valid.
```

ValidationMessages.properties文件中每条信息的key值对应于注解中message属性占位符的值。同时，最小和最大长度没有硬编码在ValidationMessages.properties文件中，在这个用户友好的信息中也有自己的占位符——{min}和{max}——它们会引用@Size注解上所设置的min和max属性。

当用户提交的注册表单校验失败的话，他们在浏览器中应该可以看到图6.3所示的界面。

将这些错误信息抽取到属性文件中还会带来一个好处，那就是我们可以通过创建地域相关的属性文件，为用户展现特定语言和地域的信息。例如，如果用户的浏览器设置成了西班牙语，那么就应该用西班牙语展现错误信息，我们需要创建一个名为ValidationErrors_es.properties的文件，内容如下：

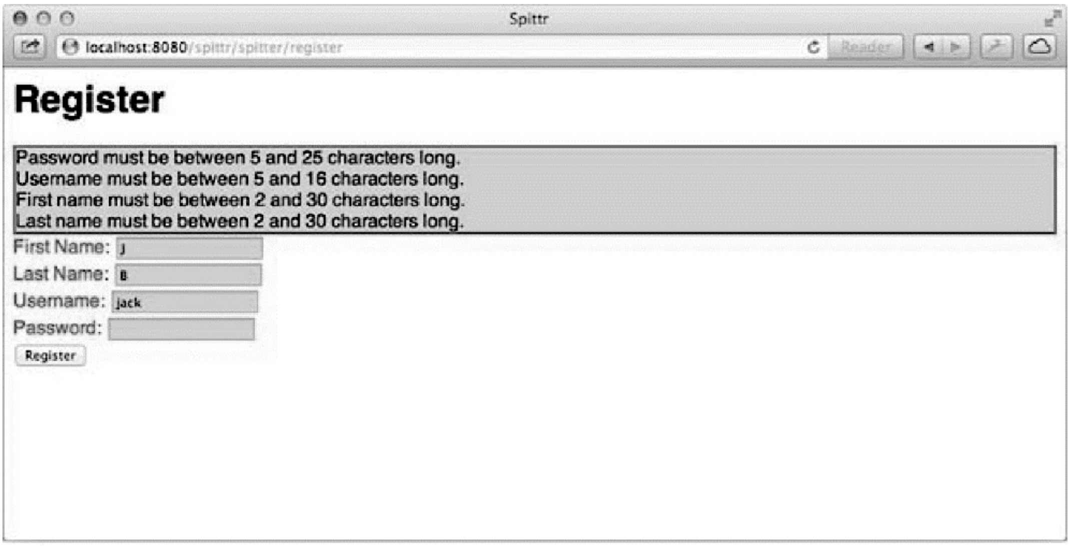

图6.3　显示校验错误，其中这些对用户友好的信息是从属性文件中获取到的

```
firstName.size=
    Nombre debe ser entre {min} y {max} caracteres largo.
lastName.size=
    El apellido debe ser entre {min} y {max} caracteres largo.
username.size=
    Nombre de usuario debe ser entre {min} y {max} caracteres largo.
password.size=
    Contraseña debe estar entre {min} y {max} caracteres largo.
email.valid=La dirección de email no es válida
```

我们可以按需创建任意数量的ValidationMessages.properties文件，使其涵盖我们想支持的所有语言和地域。

Spring通用的标签库

除了表单绑定标签库之外，Spring还提供了更为通用的JSP标签库。实际上，这个标签库是Spring中最早的标签库。这么多年来，它有所变化，但是在最早版本的Spring中，它就已经存在了。

要使用Spring通用的标签库，我们必须要在页面上对其进行声明：

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
```

与其他JSP标签库一样，`prefix`可以是任意你所喜欢的值。在这里，通用的做法是将这个标签库的前缀设置为`spring`。但是，我将其设置为“s”，因为它更加简洁，更易于阅读和输入。

标签库声明之后，我们就可以使用表6.3中的十个JSP标签了。

表**6.3　Spring**的**JSP**标签库中提供了多个便利的标签，还包括一些遗留的数据绑定标签

JSP标签	描　　述
`<s:bind>`	将绑定属性的状态导出到一个名为`status`的页面作用域属性中，与`<s:path>`组合使用获取绑定属性的值
`<s:escapeBody>`	将标签体中的内容进行HTML和/或JavaScript转义
`<s:hasBindErrors>`	根据指定模型对象（在请求属性中）是否有绑定错误，有条件地渲染内容
`<s:htmlEscape>`	为当前页面设置默认的HTML转义值

<s:message>	根据给定的编码获取信息，然后要么进行渲染（默认行为），要么将其设置为页面作用域、请求作用域、会话作用域或应用作用域的变量（通过使用var和scope属性实现）
<s:nestedPath>	设置嵌入式的path，用于<s:bind>之中
<s:theme>	根据给定的编码获取主题信息，然后要么进行渲染（默认行为），要么将其设置为页面作用域、请求作用域、会话作用域或应用作用域的变量（通过使用var和scope属性实现）
<s:transform>	使用命令对象的属性编辑器转换命令对象中不包含的属性
<s:url>	创建相对于上下文的URL，支持URI模板变量以及HTML/XML/JavaScript转义。可以渲染URL（默认行为），也可以将其设置为页面作用域、请求作用域、会话作用域或应用作用域的变量（通过使用var和scope属性实现）
<s:eval>	计算符合Spring表达式语言（Spring Expression Language，SpEL）语法的某个表达式的值，然后要么进行渲染（默认行为），要么将其设置为页面作用域、请求作用域、会话作用域或应用作用域的变量（通过使用var和scope属性实现）

表6.3中的一些标签已经被Spring表单绑定标签库淘汰了。例如，<s:bind>标签就是Spring最初所提供的表单绑定标签，它比我们在前面所介绍的标签复杂得多。

因为这些标签库的行为比表单绑定标签少得多，所以我不会详细介绍每个标签，而是快速介绍几个最为有用的标签，其余的留给读者自行去学习和探索。（即便你们会用到它们，很可能也不会那么频繁。）

展现国际化信息

到现在为止，我们的JSP模板包含了很多硬编码的文本。这其实也算不上什么大问题，但是如果你要修改这些文本的话，就不那么容易了。而且，没有办法根据用户的语言设置国际化这些文本。

例如，考虑首页中的欢迎信息：

```
<h1>Welcome to Spittr!</h1>
```

修改这个信息的唯一办法是打开home.jsp，然后对其进行变更。我觉得，这算不上什么大事。但是，应用中的文本散布到多个模板中，如果要大规模修改应用的信息时，你需要修改大量的JSP文件。

另外一个更为重要的问题在于，不管你选择什么样的欢迎信息，所有的用户都会看到同样的信息。Web是全球性的网络，你所构建的应用很可能会有全球化用户。因此，最好能够使用用户的语言与其进行交流，而不是只使用某一种语言。

对于渲染文本来说，是很好的方案，文本能够位于一个或多个属性文件中。借助<s:message>，我们可以将硬编码的欢迎信息替换为如下的形式：

```
<h1><s:message code="spittr.welcome" /></h1>
```

按照这里的方式，<s:message>将会根据key为spittr.welcome的信息源来渲染文本。因此，如果我们希望<s:message>能够正常完成任务的话，就需要配置一个这样的信息源。

Spring有多个信息源的类，它们都实现了MessageSource接口。在这些类中，更为常见和有用的是ResourceBundleMessageSource。它会从一个属性文件中加载信息，这个属性文件的名称是根据基础名称（base name）衍生而来的。如下的@Bean方法配置了ResourceBundleMessageSource：

```
@Bean
public MessageSource messageSource() {
  ResourceBundleMessageSource messageSource =
          new ResourceBundleMessageSource();
  messageSource.setBasename("messages");
  return messageSource;
}
```

在这个bean声明中，核心在于设置basename属性。你可以将其设置为任意你喜欢的值，在这里，我将其设置为message。将其设置为message后，ResourceBundle-MessageSource就会试图在根路径的属性文件中解析信息，这些属性文件的名称是根据这个基础名称衍生得到的。

另外的可选方案是使用ReloadableResourceBundleMessageSource，它的工作方

式与ResourceBundleMessageSource非常类似，但是它能够重新加载信息属性，而不必重新编译或重启应用。如下是配置ReloadableResourceBundle-MessageSource的样例：

```
@Bean
public MessageSource messageSource() {
  ReloadableResourceBundleMessageSource messageSource =
      new ReloadableResourceBundleMessageSource();
  messageSource.setBasename("file:///etc/spittr/messages");
  messageSource.setCacheSeconds(10);
  return messageSource;
}
```

这里的关键区别在于basename属性设置为在应用的外部查找（而不是像ResourceBundleMessageSource那样在类路径下查找）。basename属性可以设置为在类路径下（以“classpath:”作为前缀）、文件系统中（以“file:”作为前缀）或Web应用的根路径下（没有前缀）查找属性。在这里，我将其配置为在服务器文件系统的“/etc/spittr”目录下的属性文件中查找信息，并且基础的文件名为“message”。

现在，我们来创建这些属性文件。首先，创建默认的属性文件，名为messages. properties。它要么位于根类路径下（如果使用ResourceBundleMessageSource的话），要么位于pathname属性指定的路径下（如果使用ReloadableResourceBundle-MessageSource的话）。对spittr.welcome信息来讲，它需要如下的条目：

```
spittr.welcome=Welcome to Spittr!
```

如果你不再创建其他信息文件的话，那么我们所做的事情就是将JSP中硬编码的信息抽取到了属性文件中，依然作为硬编码的信息。它能够让我们一站式地修改应用中的所有信息，但是它所完成的任务并不限于此。

我们已经具备了对信息进行国际化的重要组成部分。例如，如果你想要为语言设置为西班牙语的用户展现西班牙语的欢迎信息，那么需要创建另外一个名为messages_es. properties的属性文件，并包含如下的条目：

```
spittr.welcome=Bienvenidos a Spittr!
```

现在，我们已经完成了一件了不起的事情。我们的应用目前只是多了几个<s:message>标签以及语言相关的属性文件，还没有完全实现

国际化！我将应用其他部分的国际化留给读者去完成。

创建**URL**

<s:url>是一个很小的标签。它主要的任务就是创建URL，然后将其赋值给一个变量或者渲染到响应中。它是JSTL中<c:url>标签的替代者，但是它具备几项特殊的技巧。

按照其最简单的形式，<s:url>会接受一个相对于Servlet上下文的URL，并在渲染的时候，预先添加上Servlet上下文路径。例如，考虑如下<s:url>的基本用法：

```
<a href="<s:url href="/spitter/register" />">Register</a>
```

如果应用的Servlet上下文名为spittr，那么在响应中将会渲染如下的HTML：

```
<a href="/spittr/spitter/register">Register</a>
```

这样，我们在创建URL的时候，就不必再担心Servlet上下文路径是什么了，<s:url>将会负责这件事。

另外，我们还可以使用<s:url>创建URL，并将其赋值给一个变量供模板在稍后使用：

```
<s:url href="/spitter/register" var="registerUrl" />
<a href="${registerUrl}">Register</a>
```

默认情况下，URL是在页面作用域内创建的。但是通过设置scope属性，我们可以让<s:url>在应用作用域内、会话作用域内或请求作用域内创建URL：

```
<s:url href="/spitter/register" var="registerUrl" scope="request" />
```

如果希望在URL上添加参数的话，那么你可以使用<s:param>标签。比如，如下的<s:url>使用两个内嵌的<s:param>标签，来设置"/spittles"的max和count参数：

```
<s:url href="/spittles" var="spittlesUrl">
  <s:param name="max" value="60" />
  <s:param name="count" value="20" />
</s:url>
```

到目前为止，我们还没有看到<s:url>能够实现，而JSTL的

<c:url>无法实现的功能。但是，如果我们需要创建带有路径（path）参数的URL该怎么办呢？我们该如何设置href属性，使其具有路径变量的占位符呢？

例如，假设我们需要为特定用户的基本信息页面创建一个URL。那没有问题，<s:param>标签可以承担此任：

```
<s:url href="/spitter/{username}" var="spitterUrl">
  <s:param name="username" value="jbauer" />
</s:url>
```

当href属性中的占位符匹配<s:param>中所指定的参数时，这个参数将会插入到占位符的位置中。如果<s:param>参数无法匹配href中的任何占位符，那么这个参数将会作为查询参数。

<s:url>标签还可以解决URL的转义需求。例如，如果你希望将渲染得到的URL内容展现在Web页面上（而不是作为超链接），那么你应该要求<s:url>进行HTML转义，这需要将htmlEscape属性设置为true。例如，如下的<s:url>将会渲染HTML转义后的URL：

```
<s:url value="/spittles" htmlEscape="true">
  <s:param name="max" value="60" />
  <s:param name="count" value="20" />
</s:url>
```

所渲染的URL结果如下所示：

```
/spitter/spittles?max=60&amp;count=20
```

另一方面，如果你希望在JavaScript代码中使用URL的话，那么应该将javaScript-Escape属性设置为true：

```
<s:url value="/spittles" var="spittlesJSUrl" javaScriptEscape="true">
  <s:param name="max" value="60" />
  <s:param name="count" value="20" />
</s:url>
<script>
  var spittlesUrl = "${spittlesJSUrl}"
</script>
```

这会渲染如下的结果到响应之中：

```
<script>
  var spittlesUrl = "\/spitter\/spittles?max=60&count=20"
</script>
```

既然提到了转义，有一个标签专门用来转义内容，而不是转义标签。接下来，让我们看一下。

转义内容

<s:escapeBody>标签是一个通用的转义标签。它会渲染标签体中内嵌的内容，并且在必要的时候进行转义。

例如，假设你希望在页面上展现一个HTML代码片段。为了正确显示，我们需要将“<”和“>”字符替换为“<”和“>”，否则的话，浏览器将会像解析页面上其他HTML那样解析这段HTML内容。

当然，没有人禁止我们手动将其转义为“<”和“>”，但是这很烦琐，并且代码难以阅读。我们可以使用<s:escapeBody>，并让Spring完成这项任务：

```
<s:escapeBody htmlEscape="true">
<h1>Hello</h1>
</s:escapeBody>
```

它将会在响应体中渲染成如下的内容：

```
&lt;h1&gt;Hello&lt;/h1&gt;
```

虽然转义后的格式看起来很难读，但浏览器会很乐意将其转换为未转义的HTML，也就是我们希望用户能够看到的样子。

通过设置javaScriptEscape属性，<s:escapeBody>标签还支持JavaScript转义：

```
<s:escapeBody javaScriptEscape="true">
<h1>Hello</h1>
</s:escapeBody>
```

<s:escapeBody>只完成一件事，并且完成得非常好。与<s:url>不同，它只会渲染内容，并不能将内容设置为变量。

现在，我们已经看到了如何使用JSP来定义Spring视图，现在让我们考虑一下如何使其在审美上更加有吸引力。我们可以在页面上增加一些通用的元素，比如添加包含站点Logo的头部、使用样式并在底部展现版权信息。我们不会在Spittr应用中的每个JSP都进行这样的修改，而是借助Apache Tiles来为模板实现一些通用且可重用的布局。

6.3 使用Apache Tiles视图定义布局

到现在为止，我们很少关心应用中Web页面的布局问题。每个JSP完全负责定义自身的布局，在这方面其实这些JSP也没有做太多工作。

假设我们想为应用中的所有页面定义一个通用的头部和底部。最原始的方式就是查找每个JSP模板，并为其添加头部和底部的HTML。但是这种方法的扩展性并不好，也难以维护。为每个页面添加这些元素会有一些初始成本，而后续的每次变更都会耗费类似的成本。

更好的方式是使用布局引擎，如Apache Tiles，定义适用于所有页面的通用页面布局。Spring MVC以视图解析器的形式为Apache Tiles提供了支持，这个视图解析器能够将逻辑视图名解析为Tile定义。

6.3.1 配置Tiles视图解析器

为了在Spring中使用Tiles，需要配置几个bean。我们需要一个`TilesConfigurer` bean，它会负责定位和加载Tile定义并协调生成Tiles。除此之外，还需要`TilesViewResolver` bean将逻辑视图名称解析为Tile定义。

这两个组件又有两种形式：针对Apache Tiles 2和Apache Tiles 3分别都有这么两个组件。这两组Tiles组件之间最为明显的区别在于包名。针对Apache Tiles 2的`TilesConfigurer`/`TilesViewResolver`位于`org.springframework.web.servlet.view.tiles2`包中，而针对Tiles 3的组件位于`org.springframework.web.servlet.view.tiles3`包中。对于该例子来讲，假设我们使用的是Tiles 3。

首先，配置`TilesConfigurer`来解析Tile定义。

程序清单**6.1** 配置**TilesConfigurer**来解析定义

```
@Bean
public TilesConfigurer tilesConfigurer() {
  TilesConfigurer tiles = new TilesConfigurer();
  tiles.setDefinitions(new String[] {          <-- 指定 Tile 定义的位置
      "/WEB-INF/layout/tiles.xml"
  });
  tiles.setCheckRefresh(true);         <-- 启用刷新功能
  return tiles;
}
```

当配置`TilesConfigurer`的时候，所要设置的最重要的属性就是`definitions`。这个属性接受一个`String`类型的数组，其中每个条目都指定一个Tile定义的XML文件。对于Spittr应用来讲，我们让它在“/WEB-INF/layout/”目录下查找tiles.xml。

其实我们还可以指定多个Tile定义文件，甚至能够在路径位置上使用通配符，当然在上例中我们没有使用该功能。例如，我们要求TilesConfigurer加载“/WEB-INF/”目录下的所有名字为tiles.xml的文件，那么可以按照如下的方式设置definitions属性：

```
tiles.setDefinitions(new String[] {
    "/WEB-INF/**/tiles.xml"
});
```

在本例中，我们使用了Ant风格的通配符（**），所以TilesConfigurer会遍历“WEB-INF/”的所有子目录来查找Tile定义。

接下来，让我们来配置TilesViewResolver，可以看到，这是一个很基本的bean定义，没有什么要设置的属性：

```
@Bean
public ViewResolver viewResolver() {
  return new TilesViewResolver();
}
```

如果你更喜欢XML配置的话，那么可以按照如下的形式配置TilesConfigurer和TilesViewResolver：

```
<bean id="tilesConfigurer" class=
    "org.springframework.web.servlet.view.tiles3.TilesConfigurer">
  <property name="definitions">
    <list>
      <value>/WEB-INF/layout/tiles.xml.xml</value>
      <value>/WEB-INF/views/**/tiles.xml</value>
    </list>
  </property>
</bean>

<bean id="viewResolver" class=
    "org.springframework.web.servlet.view.tiles3.TilesViewResolver" />
```

TilesConfigurer会加载Tile定义并与Apache Tiles协作，而TilesViewRe-solver会将逻辑视图名称解析为引用Tile定义的视图。它是通过查找与逻辑视图名称相匹配的Tile定义实现该功能的。我们需要创建几个Tile定义以了解它是如何运转的。

定义Tiles

Apache Tiles提供了一个文档类型定义（document type definition，DTD），用来在XML文件中指定Tile的定义。每个定义中需要包含一个<definition>元素，这个元素会有一个或多个<put-attribute>元素。例如，如下的XML文档为Spittr应用定义了几个

Tile。

程序清单**6.2**　为**Spittr**应用定义**Tile**

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
       "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>

  <definition name="base"                                  <— 定义 base Tile
              template="/WEB-INF/layout/page.jsp">

  <put-attribute name="header"
                 value="/WEB-INF/layout/header.jsp" />
  <put-attribute name="footer"
                 value="/WEB-INF/layout/footer.jsp" />    <— 设置属性
</definition>

<definition name="home" extends="base">                   <— 扩展 base Tile
  <put-attribute name="body"
                 value="/WEB-INF/views/home.jsp" />
</definition>

<definition name="registerForm" extends="base">
  <put-attribute name="body"
                 value="/WEB-INF/views/registerForm.jsp" />
</definition>

<definition name="profile" extends="base">
    <put-attribute name="body"
                   value="/WEB-INF/views/profile.jsp" />
  </definition>

  <definition name="spittles" extends="base">
    <put-attribute name="body"
                   value="/WEB-INF/views/spittles.jsp" />
  </definition>

  <definition name="spittle" extends="base">
    <put-attribute name="body"
                   value="/WEB-INF/views/spittle.jsp" />
  </definition>

</tiles-definitions>
```

每个`<definition>`元素都定义了一个Tile，它最终引用的是一个JSP模板。在名为`base`的Tile中，模板引用的是“/WEB-INF/layout/page.jsp”。某个Tile可能还会引用其他的JSP模板，使这些JSP模板嵌入到主模板中。对于base Tile来讲，它引用的是一个头部JSP模板和一个底部JSP模板。

base Tile所引用的page.jsp模板如下面程序清单所示。

程序清单**6.3**　主布局模板：引用其他模板来创建视图

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="t" %>
<%@ page session="false" %>
<html>
  <head>
    <title>Spittr</title>
    <link rel="stylesheet"
          type="text/css"
          href="<s:url value="/resources/style.css" />" >
  </head>
  <body>
    <div id="header">

      <t:insertAttribute name="header" />          <— 插入头部
    </div>

    <div id="content">
      <t:insertAttribute name="body" />          <— 插入主体内容
    </div>
    <div id="footer">
      <t:insertAttribute name="footer" />          <— 插入底部
    </div>
  </body>
</html>
```

在程序清单6.3中，需要重点关注的事情就是如何使用Tile标签库中的`<t:insert Attribute>` JSP标签来插入其他的模板。在这里，用它来插入名为`header`、`body`和`footer`的模板。最终，它会形成图6.4所示的布局。

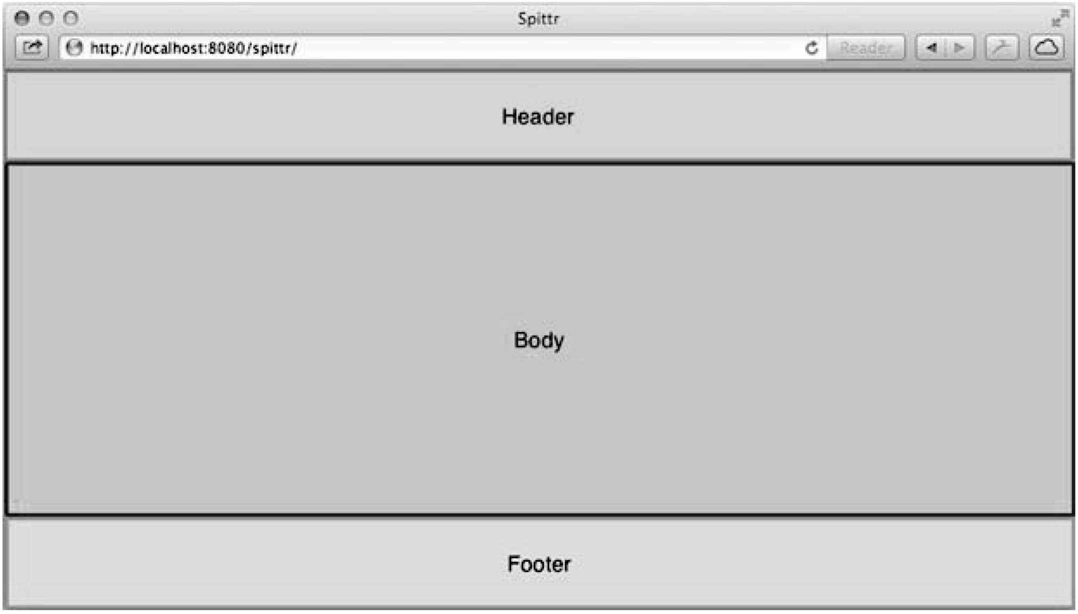

图6.4　通用的布局，定义了头部、主体区以及底部

在base Tile定义中，`header`和`footer`属性分别被设置为引用“/WEB-INF/layout/ header. jsp”和“/WEB-INF/layout/footer.jsp”。但是body属性呢？它是在哪里设置的呢？

在这里，`base` Tile不会期望单独使用。它会作为基础定义（这是其

名字的来历），供其他的Tile定义扩展。在程序清单6.2的其余内容中，我们可以看到其他的Tile定义都是扩展自base Tile。它意味着它们会继承其header和footer属性的设置（当然，Tile定义中也可以覆盖掉这些属性），但是每一个都设置了body属性，用来指定每个Tile特有的JSP模板。

现在，我们关注一下home Tile，它扩展了base。因为它扩展了base，因此它会继承base中的模板和所有的属性。尽管home Tile定义相对来说很简单，但是它实际上包含了如下的定义：

```
<definition name="home" template="/WEB-INF/layout/page.jsp">
  <put-attribute name="header" value="/WEB-INF/layout/header.jsp" />
  <put-attribute name="footer" value="/WEB-INF/layout/footer.jsp" />
  <put-attribute name="body" value="/WEB-INF/views/home.jsp" />
</definition>
```

属性所引用的每个模板是很简单的，如下是header.jsp模板：

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
<a href="<s:url value="/" />"><img
    src="<s:url value="/resources" />/images/spittr_logo_50.png"
    border="0"/></a>
```

footer.jsp模板更为简单：

```
Copyright &copy; Craig Walls
```

每个扩展自base的Tile都定义了自己的主体区模板，所以每个都会与其他的有所区别。但是为了完整地了解home Tile，如下展现了home.jsp：

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<h1>Welcome to Spittr</h1>

<a href="<c:url value="/spittles" />">Spittles</a> |
<a href="<c:url value="/spitter/register" />">Register</a>
```

这里的关键点在于通用的元素放到了page.jsp、header.jsp以及footer.jsp中，其他的Tile模板中不再包含这部分内容。这使得它们能够跨页面重用，这些元素的维护也得以简化。

要想看一下这些元素组合在一起的样子，那么可以看一下图6.5。如图所示，它包含了一些样式和图像以增加应用的美观性。我们不是专门讨论使用Tiles实现页面布局的，因此在本节中不会涵盖所有的细节。但是，我们可以看到页面上的各种组件通过Tile定义组合在了一

起，并且渲染出了Spittr应用的主页。

在Java Web应用领域，JSP长期以来都是占据主导地位的方案。但是，在这个领域有了新的竞争者，也就是Thymeleaf。接下来让我们看一下如何在Spring MVC应用中使用Thymeleaf。

图6.5　Spittr首页，通过Apache Tiles进行的布局

6.4　使用Thymeleaf

尽管JSP已经存在了很长的时间，并且在Java Web服务器中无处不在，但是它却存在一些缺陷。JSP最明显的问题在于它看起来像HTML或XML，但它其实上并不是。大多数的JSP模板都是采用HTML的形式，但是又掺杂上了各种JSP标签库的标签，使其变得很混乱。这些标签库能够以很便利的方式为JSP带来动态渲染的强大功能，但是它也摧毁了我们想维持一个格式良好的文档的可能性。作为一个极端的样例，如下的JSP标签甚至作为HTML参数的值：

```
<input type="text" value="<c:out value="${thing.name}"/>" />
```

标签库和JSP缺乏良好格式的一个副作用就是它很少能够与其产生的HTML类似。所以，在Web浏览器或HTML编辑器中查看未经渲染的JSP模板是非常令人困惑的，而且得到的结果看上去也非常丑陋。这个结果是不完整的——在视觉上这简直就是一场灾难！因为JSP并不

是真正的HTML，很多浏览器和编辑器展现的效果都很难在审美上接近模板最终所渲染出来的效果。

同时，JSP规范是与Servlet规范紧密耦合的。这意味着它只能用在基于Servlet的Web应用之中。JSP模板不能作为通用的模板（如格式化Email），也不能用于非Servlet的Web应用。

多年来，在Java应用中，有多个项目试图挑战JSP在视图领域的统治性地位。最新的挑战者是Thymeleaf，它展现了一些切实的承诺，是一项很令人兴奋的可选方案。Thymeleaf模板是原生的，不依赖于标签库。它能在接受原始HTML的地方进行编辑和渲染。因为它没有与Servlet规范耦合，因此Thymeleaf模板能够进入JSP所无法涉足的领域。现在，我们看一下如何在Spring MVC中使用Thymeleaf。

6.4.1 配置Thymeleaf视图解析器

为了要在Spring中使用Thymeleaf，我们需要配置三个启用Thymeleaf与Spring集成的bean：

- `ThymeleafViewResolver`：将逻辑视图名称解析为Thymeleaf模板视图；
- `SpringTemplateEngine`：处理模板并渲染结果；
- `TemplateResolver`：加载Thymeleaf模板。

如下为声明这些bean的Java配置。

程序清单**6.4**　使用**Java**代码的方式，配置**Spring**对**Thymeleaf**的支持

```
@Bean
public ViewResolver viewResolver(                    <—Thymeleaf 视图解析器
      SpringTemplateEngine templateEngine) {
  ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
  viewResolver.setTemplateEngine(templateEngine);
  return viewResolver;
}

@Bean
public TemplateEngine templateEngine(                     <—模板引擎
      TemplateResolver templateResolver) {
  SpringTemplateEngine templateEngine = new SpringTemplateEngine();
  templateEngine.setTemplateResolver(templateResolver);
  return templateEngine;
}

@Bean
public TemplateResolver templateResolver() {           <— 模板解析器
  TemplateResolver templateResolver =
         new ServletContextTemplateResolver();
  templateResolver.setPrefix("/WEB-INF/templates/");
  templateResolver.setSuffix(".html");
  templateResolver.setTemplateMode("HTML5");
  return templateResolver;
}
```

如果你更愿意使用XML来配置bean，那么如下的`<bean>`声明能够完成该任务。

程序清单**6.5**　使用**XML**的方式，配置**Spring**对**Thymeleaf**的支持

```
<bean id="viewResolver"                           <— Thymeleaf 视图解析器
      class="org.thymeleaf.spring3.view.ThymeleafViewResolver"
      p:templateEngine-ref="templateEngine" />

<bean id="templateEngine"                          <—  模板引擎
      class="org.thymeleaf.spring3.SpringTemplateEngine"
      p:templateResolver-ref="templateResolver" />

<bean id="templateResolver" class=                  <— 模板解析器
    "org.thymeleaf.templateresolver.ServletContextTemplateResolver"
      p:prefix="/WEB-INF/templates/"
      p:suffix=".html"
      p:templateMode="HTML5" />
```

不管使用哪种配置方式，Thymeleaf都已经准备就绪了，它可以将响应中的模板渲染到Spring MVC控制器所处理的请求中。

`ThymeleafViewResolver`是Spring MVC中`ViewResolver`的一个实现类。像其他的视图解析器一样，它会接受一个逻辑视图名称，并将其解析为视图。不过在该场景下，视图会是一个Thymeleaf模板。

需要注意的是`ThymeleafViewResolver` bean中注入了一个对`SpringTemplate Engine` bean的引用。`SpringTemplateEngine`会在Spring中启用Thymeleaf引擎，用来解析模板，并基于这些模板渲染结果。可以看到，我们为其注入了一个`TemplateResolver` bean的引用。

TemplateResolver会最终定位和查找模板。与之前配置InternalResource-ViewResolver类似，它使用了prefix和suffix属性。前缀和后缀将会与逻辑视图名组合使用，进而定位Thymeleaf引擎。它的templateMode属性被设置成了HTML 5，这表明我们预期要解析的模板会渲染成HTML 5输出。

所有的Thymeleaf bean都已经配置完成了，那么接下来我们该创建几个视图了。

6.4.2　定义Thymeleaf模板

Thymeleaf在很大程度上就是HTML文件，与JSP不同，它没有什么特殊的标签或标签库。Thymeleaf之所以能够发挥作用，是因为它通过自定义的命名空间，为标准的HTML标签集合添加Thymeleaf属性。如下的程序清单展现了home.html，也就是使用Thymeleaf命名空间的首页模板。

程序清单**6.6　home.html**：使用**Thymeleaf**命名空间的首页模板引擎

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">        <—— 声明 Thymeleaf 命名空间
  <head>
    <title>Spittr</title>
    <link rel="stylesheet"
          type="text/css"
          th:href="@{/resources/style.css}"></link>  <—— 到样式表的 th:href 链接
  </head>
  <body>
    <h1>Welcome to Spittr</h1>

    <a th:href="@{/spittles}">Spittles</a> |        <—— 到页面的 th:href 链接
    <a th:href="@{/spitter/register}">Register</a>
  </body>
</html>
```

首页模板相对来讲很简单，只使用了th:href属性。这个属性与对应的原生HTML属性很类似，也就是href属性，并且可以按照相同的方式来使用。th:href属性的特殊之处在于它的值中可以包含Thymeleaf表达式，用来计算动态的值。它会渲染成一个标准的href属性，其中会包含在渲染时动态创建得到的值。这是Thymeleaf命名空间中很多属性的运行方式：它们对应标准的HTML属性，并且具有相同的名称，但是会渲染一些计算后得到的值。在本例中，使用th:href属性的三个地方都用到了“@{}”表达式，用来计算相对于URL的路径（就像在JSP页面中，我们可能会使用的JSTL <c:url>标签或Spring<s:url>标签类似）。

尽管home.html是一个相当简单的Thymeleaf模板，但是它依然很有价

值，这在于它与纯HTML模板非常接近。唯一的区别之处在于th:href属性，否则的话，它就是基础且功能丰富的HTML文件。

这意味着Thymeleaf模板与JSP不同，它能够按照原始的方式进行编辑甚至渲染，而不必经过任何类型的处理器。当然，我们需要Thymeleaf来处理模板并渲染得到最终期望的输出。即便如此，如果没有任何特殊的处理，home.html也能够加载到Web浏览器中，并且看上去与完整渲染的效果很类似。为了更加清晰地阐述这一点，图6.6对比了home.jsp（上方）和home.html（下方）在Web浏览器中的显式效果。

可以看到，在Web浏览器中，JSP模板的渲染效果很糟糕。尽管我们可以看到一些熟悉的元素，但是JSP标签库的声明也显示了出来。在链接前出现了一些令人费解的未闭合标记，这是Web浏览器没有正常解析<s:url>标签的结果。

与之相反，Thymeleaf模板的渲染效果基本上没有任何错误。稍微有点问题的是链接部分，Web浏览器并不会像处理href属性那样处理th:href，所以链接并没有渲染为链接的样子。除了这些细微的问题，模板的渲染效果与我们的预期完全符合。

像home.jsp这样的模板作为Thymeleaf入门是很合适的。但是Spring的JSP标签所擅长的是表单绑定。如果我们抛弃JSP的话，那是不是也要抛弃表单绑定呢？不必担心。Thymeleaf提供了与之相匹敌的功能。

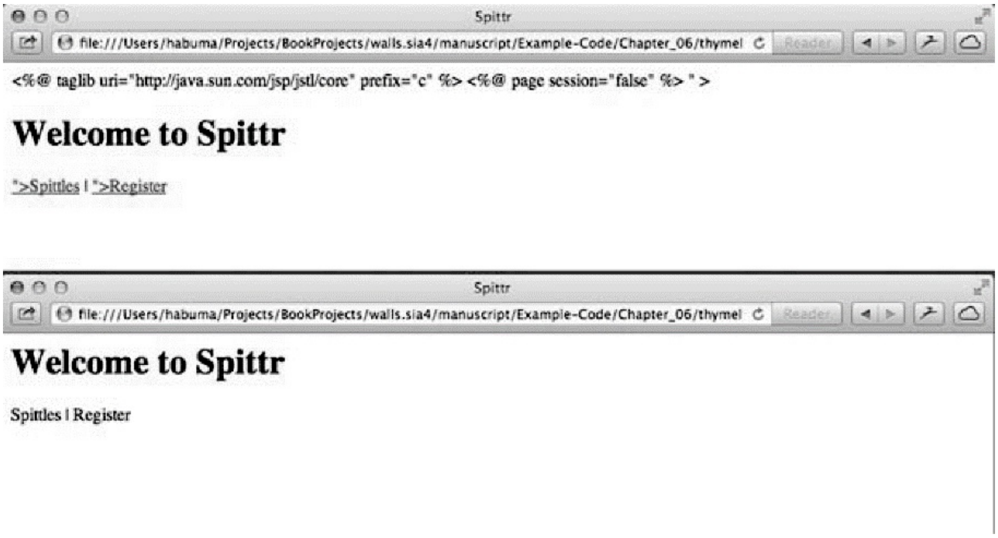

图6.6　Thymeleaf模板与JSP不同，它是HTML，可以像HTML那样进行渲染和编辑

借助**Thymeleaf**实现表单绑定

表单绑定是Spring MVC的一项重要特性。它能够将表单提交的数据填充到命令对象中，并将其传递给控制器，而在展现表单的时候，表单中也会填充命令对象中的值。如果没有表单绑定功能的话，我们需要确保HTML表单域要映射后端命令对象中的属性，并且在校验失败后展现表单的时候，还要负责确保输入域中值要设置为命令对象的属性。

但是，如果有表单绑定的话，它就会负责这些事情了。为了复习一下表单绑定是如何运行的，下面展现了在registration.jsp中的First Name输入域：

```
<sf:label path="firstName"
    cssErrorClass="error">First Name</sf:label>:
  <sf:input path="firstName" cssErrorClass="error" /><br/>
```

在这里，调用了Spring表单绑定标签库的`<sf:input>`标签，它会渲染出一个HTML `<input>`标签，并且其`value`属性设置为后端对象`firstName`属性的值。它还使用了Spring的`<sf:label>`标签及其`cssErrorClass`属性，如果出现校验错误的话，会将文本标记渲染为红色。

但是，我们本节讨论的并不是JSP，而是使用Thymeleaf替换JSP。因此，我们不能使用Spring的JSP标签实现表单绑定，而是使用Thymeleaf的Spring方言。

作为阐述的样例，请参考如下的Thymeleaf模板片段，它会渲染First Name输入域：

```
<label th:class="${#fields.hasErrors('firstName')}? 'error'">
    First Name</label>:
<input type="text" th:field="*{firstName}"
       th:class="${#fields.hasErrors('firstName')}? 'error'" /><br/>
```

在这里，我们不再使用Spring JSP标签中的`cssClassName`属性，而是在标准的HTML标签上使用`th:class`属性。`th:class`属性会渲染为一个`class`属性，它的值是根据给定的表达式计算得到的。在上面的这两个`th:class`属性中，它会直接检查`firstName`域有没有校验错误。如果有的话，`class`属性在渲染时的值为`error`。如果这个域没有错误的话，将不会渲染`class`属性。

`<input>`标签使用了`th:field`属性，用来引用后端对象的`firstName`域。这可能与你的预期有点差别。在Thymeleaf模板中，我们在很多情况下所使用的属性都对应于标准的HTML属性，因此貌

似使用th:value属性来设置<input>标签的value属性才是合理的。

其实不然，因为我们是在将这个输入域绑定到后端对象的firstName属性上，因此使用th:field属性引用firstName域。通过使用th:field，我们将value属性设置为firstName的值，同时还会将name属性设置为firstName。

为了阐述Thymeleaf是如何实际运行的，如下的程序清单展示了完整的注册表单模板。

程序清单**6.7**　注册页面，使用**Thymeleaf**将一个表单绑定到命令对象上

```
<form method="POST" th:object="${spitter}">
展示错误
  <div class="errors" th:if="${#fields.hasErrors('*')}">
    <ul>
      <li th:each="err : ${#fields.errors('*')}"
          th:text="${err}">Input is incorrect</li>
    </ul>
  </div>
FirstName
  <label th:class="${#fields.hasErrors('firstName')}? 'error'">
      First Name</label>:
    <input type="text" th:field="*{firstName}"
           th:class="${#fields.hasErrors('firstName')}? 'error'" /><br/>
Last Name
  <label th:class="${#fields.hasErrors('lastName')}? 'error'">
      Last Name</label>:
    <input type="text" th:field="*{lastName}"
           th:class="${#fields.hasErrors('lastName')}? 'error'" /><br/>
Email
  <label th:class="${#fields.hasErrors('email')}? 'error'">
      Email</label>:
    <input type="text" th:field="*{email}"
           th:class="${#fields.hasErrors('email')}? 'error'" /><br/>
Username
  <label th:class="${#fields.hasErrors('username')}? 'error'">
      Username</label>:
    <input type="text" th:field="*{username}"
           th:class="${#fields.hasErrors('username')}? 'error'" /><br/>
Password
  <label th:class="${#fields.hasErrors('password')}? 'error'">
      Password</label>:
    <input type="password" th:field="*{password}"
           th:class="${#fields.hasErrors('password')}? 'error'" /><br/>

  <input type="submit" value="Register" />
</form>
```

程序清单6.7使用了相同的Thymeleaf属性和“*{}”表达式，为所有的表单域绑定后端对象。这其实重复了我们在First Name域中所做的事情。

但是，需要注意我们在表单的顶部了也使用了Thymeleaf，它会用来渲染所有的错误。<div>元素使用th:if属性来检查是否有校验错误。如果有的话，会渲染<div>，否则的话，它将不会渲染。

在<div>中，会使用一个无顺序的列表来展现每项错误。<li>标签上的th:each属性将会通知Thymeleaf为每项错误都渲染一个<li>，在每次迭代中会将当前错误设置到一个名为err的变量中。

标签还有一个th:text属性。这个命令会通知Thymeleaf计算某一个表达式（在本例中，也就是err变量）并将它的值渲染为<li>标签的内容体。实际上的效果就是每项错误对应一个<li>元素，并展现错误的文本。

你可能会想知道“${}”和“*{}”括起来的表达式到底有什么区别。“${}”表达式（如${spitter}）是变量表达式（variable expression）。一般来讲，它们会是对象图导航语言（Object-Graph Navigation Language，OGNL）表达式（http://commons.apache.org/proper/commons-ognl/）。但在使用Spring的时候，它们是SpEL表达式。在${spitter}这个例子中，它会解析为key为spitter的model属性。

而对于“*{}”表达式，它们是选择表达式（selection expression）。变量表达式是基于整个SpEL上下文计算的，而选择表达式是基于某一个选中对象计算的。在本例的表单中，选中对象就是<form>标签中th:object属性所设置的对象：模型中的Spitter对象。因此，“*{firstName}”表达式就会计算为Spitter对象的firstName属性。

6.5　小结

处理请求只是Spring MVC功能的一部分。如果控制器所产生的结果想要让人看到，那么它们产生的模型数据就要渲染到视图中，并展现到用户的Web浏览器中。Spring的视图渲染是很灵活的，并提供了多个内置的可选方案，包括传统的JavaServer Pages以及流行的Apache Tiles布局引擎。

在本章中，我们首先快速了解了一下Spring所提供的视图和视图解析可选方案。我们还深入学习了如何在Spring MVC中使用JSP和Apache Tiles。

我们还看到了如何使用Thymeleaf作为Spring MVC应用的视图层，它被视为JSP的替代方案。Thymeleaf是一项很有吸引力的技术，因为它能创建原始的模板，这些模板是纯HTML，能像静态HTML那样以原

始的方式编写和预览，并且能够在运行时渲染动态模型数据。除此之外，Thymeleaf是与Servlet没有耦合关系的，这样它就能够用在JSP所不能使用的领域中。

Spittr应用的视图定义完成之后，我们已经具有了一个虽然微小但是可部署且具有一定功能的Spring MVC Web应用。还有一些其他的特性需要更新进来，如数据持久化和安全性，我们会在合适的时候关注这些特性。但现在，这个应用开始变得有模有样了。

在深入学习应用的技术栈之前，在下一章我们将会继续讨论Spring MVC，学习这个框架中一些更为有用和高级的功能。

第7章　Spring MVC的高级技术

本章内容：

- Spring MVC配置的替代方案
- 处理文件上传
- 在控制器中处理异常
- 使用flash属性

稍等，还没有结束！

如果你在电视购物节目上看过一些小发明或产品广告的话，你可能听过类似这样的话。在广告描述完产品并宣称它能够做什么之后，我们可能会听到“稍等，还没有结束！”，然后广告会继续告诉我们产品还有什么令人激动的特性。

在很多方面，Spring MVC（其实，整个Spring也是如此）也有“还没有结束！”这样的感觉。就在我们觉得已经掌握了Spring MVC能够做什么之后，我们会发现它所能做的还不止如此。

在第5章中，我们学习了Spring MVC的基础知识，以及如何编写控制器来处理各种请求。基于这些知识，我们在第6章学习了如何创建JSP和Thymeleaf视图，这些视图会将模型数据展现给用户。你可能认为我们已经掌握了Spring MVC的全部知识。但是稍等！还没有结束！

在本章中，我们会继续Spring MVC的话题，本章所介绍的特性已经超出了第5章和第6章基础知识的范畴。我们将会看到如何编写控制器来处理文件上传、如何处理控制器所抛出的异常，以及如何在模型中传递数据，使其能够在重定向（redirect）之后依然存活。

但首先，我要兑现一个承诺。在第5章中，我快速展现了如何通过`AbstractAnnotationConfigDispatcherServletInitializ`搭建Spring MVC，当时我承诺会为读者展现其他的配置方案。所以，在介绍文件上传和异常处理之前，我们先花一点时间探讨一下如何用其他的方式来搭建`DispatcherServlet`和`ContextLoaderListener`。

7.1　Spring MVC配置的替代方案

在第5章中，我们通过扩展AbstractAnnotationConfigDispatcherServlet-Initializer快速搭建了Spring MVC环境。在这个便利的基础类中，假设我们需要基本的DispatcherServlet和ContextLoaderListener环境，并且Spring配置是使用Java的，而不是XML。

尽管对很多Spring应用来说，这是一种安全的假设，但是并不一定总能满足我们的要求。除了DispatcherServlet以外，我们可能还需要额外的Servlet和Filter；我们可能还需要对DispatcherServlet本身做一些额外的配置；或者，如果我们需要将应用部署到Servlet 3.0之前的容器中，那么还需要将DispatcherServlet配置到传统的web.xml中。

7.1.1 自定义DispatcherServlet配置

虽然从程序清单7.1的外观上不一定能够看得出来，但是Abstract-AnnotationConfigDispatcherServletInitializer所完成的事情其实比看上去要多。在SpittrWebAppInitializer中我们所编写的三个方法仅仅是必须要重载的abstract方法。但实际上还有更多的方法可以进行重载，从而实现额外的配置。

此类的方法之一就是customizeRegistration()。在AbstractAnnotation-ConfigDispatcherServletInitializer将DispatcherServlet注册到Servlet容器中之后，就会调用customizeRegistration()，并将Servlet注册后得到的Registration.Dynamic传递进来。通过重载customizeRegistration()方法，我们可以对DispatcherServlet进行额外的配置。

例如，在本章稍后的内容中（7.2节），我们将会看到如何在Spring MVC中处理multipart请求和文件上传。如果计划使用Servlet 3.0对multipart配置的支持，那么需要使用DispatcherServlet的registration来启用multipart请求。我们可以重载customizeRegistration()方法来设置MultipartConfigElement，如下所示：

```java
@Override
protected void customizeRegistration(Dynamic registration) {
  registration.setMultipartConfig(
      new MultipartConfigElement("/tmp/spittr/uploads"));
}
```

借助`customizeRegistration()`方法中的`ServletRegistration.Dynamic`，我们能够完成多项任务，包括通过调用`setLoadOnStartup()`设置load-on-startup优先级，通过`setInitParameter()`设置初始化参数，通过调用`setMultipartConfig()`配置Servlet 3.0对multipart的支持。在前面的样例中，我们设置了对multipart的支持，将上传文件的临时存储目录设置在“/tmp/spittr/uploads”中。

7.1.2　添加其他的**Servlet**和**Filter**

按照`AbstractAnnotationConfigDispatcherServletInitiali`的定义，它会创建`DispatcherServlet`和`ContextLoaderListener`。但是，如果你想注册其他的Servlet、Filter或Listener的话，那该怎么办呢？

基于Java的初始化器（initializer）的一个好处就在于我们可以定义任意数量的初始化器类。因此，如果我们想往Web容器中注册其他组件的话，只需创建一个新的初始化器就可以了。最简单的方式就是实现Spring的`WebApplicationInitializer`接口。

例如，如下的程序清单展现了如何创建`WebApplicationInitializer`实现并注册一个Servlet。

程序清单**7.1**　通过实现**WebApplicationInitializer**来注册**Servlet**

```
package com.myapp.config;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration.Dynamic;
import org.springframework.web.WebApplicationInitializer;
import com.myapp.MyServlet;

public class MyServletInitializer implements WebApplicationInitializer {

  @Override
  public void onStartup(ServletContext servletContext)
         throws ServletException {
                                                      注册 Servlet
    Dynamic myServlet =                             <-
        servletContext.addServlet("myServlet", MyServlet.class);

    myServlet.addMapping("/custom/**");              <- 映射 Servlet
  }

}
```

程序清单7.1是相当基础的Servlet注册初始化器类。它注册了一个Servlet并将其映射到一个路径上。我们也可以通过这种方式来手动注册`DispatcherServlet`。（但这并没有必要，因为`AbstractAnnotationConfigDispatcherServletInitiali`没用太多代码就将这项任务完成得很漂亮。）

类似地，我们还可以创建新的`WebApplicationInitializer`实现来注册Listener和Filter。例如，如下的程序清单展现了如何注册Filter。

程序清单**7.2**　注册**Filter**的**WebApplicationInitializer**

```
@Override
public void onStartup(ServletContext servletContext)
       throws ServletException {
                                                           注册 Filter
  javax.servlet.FilterRegistration.Dynamic filter =
      servletContext.addFilter("myFilter", MyFilter.class);   <-

  filter.addMappingForUrlPatterns(null, false, "/custom/*");  <- 添加 Filter 的映
}                                                                射路径
```

如果要将应用部署到支持Servlet 3.0的容器中，那么`WebApplicationInitializer`提供了一种通用的方式，实现在Java中注册Servlet、Filter和Listener。不过，如果你只是注册Filter，并且该Filter只会映射到`DispatcherServlet`上的话，那么在`AbstractAnnotationConfigDispatcherServletInitiali`中还有一种快捷方式。

为了注册Filter并将其映射到`DispatcherServlet`，所需要做的仅仅是重载`AbstractAnnotationConfigDispatcherServletInitiali`

的getServlet-Filters()方法。例如，在如下的代码中，重载了AbstractAnnotationConfig-DispatcherServletInitializer的getServletFilters()方法以注册Filter：

```
@Override
protected Filter[] getServletFilters() {
  return new Filter[] { new MyFilter() };
}
```

我们可以看到，这个方法返回的是一个javax.servlet.Filter的数组。在这里它只返回了一个Filter，但它实际上可以返回任意数量的Filter。在这里没有必要声明它的映射路径，getServletFilters()方法返回的所有Filter都会映射到DispatcherServlet上。

如果要将应用部署到Servlet 3.0容器中，那么Spring提供了多种方式来注册Servlet（包括DispatcherServlet）、Filter和Listener，而不必创建web.xml文件。但是，如果你不想采取以上所述方案的话，也是可以的。假设你需要将应用部署到不支持Servlet 3.0的容器中（或者你只是希望使用web.xml文件），那么我们完全可以按照传统的方式，通过web.xml配置Spring MVC。让我们看一下该怎么做。

7.1.3 在web.xml中声明DispatcherServlet

在典型的Spring MVC应用中，我们会需要DispatcherServlet和Context-Loader Listener。AbstractAnnotationConfigDispatcherServlet会自动注册它们，但是如果需要在web.xml中注册的话，那就需要我们自己来完成这项任务了。

如下是一个基本的web.xml文件，它按照传统的方式搭建了DispatcherServlet和ContextLoaderListener。

程序清单**7.3** 在**web.xml**中搭建**Spring MVC**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

设置根上下文配置文件位置

注册 ContextLoader-Listener

注册 Dispatcher-Servlet

将 DispatcherServlet 映射到“/”

就像我在第5章曾经介绍过的，ContextLoaderListener和DispatcherServlet各自都会加载一个Spring应用上下文。上下文参数contextConfigLocation指定了一个XML文件的地址，这个文件定义了根应用上下文，它会被ContextLoaderListener加载。如程序清单7.3所示，根上下文会从“/WEB-INF/spring/root-context.xml”中加载bean定义。

DispatcherServlet会根据Servlet的名字找到一个文件，并基于该文件加载应用上下文。在程序清单7.3中，Servlet的名字是appServlet，因此DispatcherServlet会从“/WEB-INF/appServlet-context.xml”文件中加载其应用上下文。

如果你希望指定DispatcherServlet配置文件的位置的话，那么可以在Servlet上指定一个contextConfigLocation初始化参数。例如，如下的配置中，DispatcherServlet会从“/WEB-INF/spring/appServlet/servlet-context.xml”加载它的bean：

```xml
<servlet>
  <servlet-name>appServlet</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/spring/appServlet/servlet-context.xml
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

当然，上面阐述的都是如何让`DispatcherServlet`和`ContextLoaderListener`从XML中加载各自的应用上下文。但是，在本书中的大部分内容中，我们都更倾向于使用Java配置而不是XML配置。因此，我们需要让Spring MVC在启动的时候，从带有`@Configuration`注解的类上加载配置。

要在Spring MVC中使用基于Java的配置，我们需要告诉`DispatcherServlet`和`ContextLoaderListener`使用`AnnotationConfigWebApplicationContext`，这是一个`WebApplicationContext`的实现类，它会加载Java配置类，而不是使用XML。要实现这种配置，我们可以设置contextClass上下文参数以及`DispatcherServlet`的初始化参数。如下的程序清单展现了一个新的web.xml，在这个文件中，它所搭建的Spring MVC使用基于Java的Spring配置：

程序清单**7.4**　设置**web.xml**使用基于**Java**的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
                                                              使用 Java
  <context-param>                                             配置
    <param-name>contextClass</param-name>                 <--
    <param-value>
      org.springframework.web.context.support.
                              ➥ AnnotationConfigWebApplicationContext
    </param-value>
  </context-param>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>com.habuma.spitter.config.RootConfig</param-value>    <--
  </context-param>                                               指定根配置类

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>                                           使用 Java
    <init-param>                                               配置
      <param-name>contextClass</param-name>          <--
      <param-value>
        org.springframework.web.context.support.
                              ➥ AnnotationConfigWebApplicationContext
      </param-value>
    </init-param>
    <init-param>
      <param-name>contextConfigLocation</param-name>    <--   指定 DispatcherServl
      <param-value>                                           配置类
        com.habuma.spitter.config.WebConfigConfig
      </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

现在我们已经看到了如何以多种不同的方式来搭建Spring MVC，那么接下来我们会看一下如何使用Spring MVC来处理文件上传。

7.2 处理multipart形式的数据

在Web应用中，允许用户上传内容是很常见的需求。在Facebook和

Flickr这样的网站中，用户通常会上传照片和视频，并与家人和朋友分享。还有一些服务允许用户上传照片，然后按照传统方式将其打印在纸上，或者用在T恤衫和咖啡杯上。

Spittr应用在两个地方需要文件上传。当新用户注册应用的时候，我们希望他们能够上传一张图片，从而与他们的个人信息相关联。当用户提交新的`Spittle`时，除了文本消息以外，他们可能还会上传一张照片。

一般表单提交所形成的请求结果是很简单的，就是以“&”符分割的多个name-value对。例如，当在Spittr应用中提交注册表单时，请求会如下所示：

```
firstName=Charles&lastName=Xavier&email=professorx%40xmen.org
&username=professorx&password=letmein01
```

尽管这种编码形式很简单，并且对于典型的基于文本的表单提交也足够满足要求，但是对于传送二进制数据，如上传图片，就显得力不从心了。与之不同的是，multipart格式的数据会将一个表单拆分为多个部分（part），每个部分对应一个输入域。在一般的表单输入域中，它所对应的部分中会放置文本型数据，但是如果上传文件的话，它所对应的部分可以是二进制，下面展现了multipart的请求体：

```
------WebKitFormBoundaryqgkaBn8IHJCuNmiW
Content-Disposition: form-data; name="firstName"

Charles
------WebKitFormBoundaryqgkaBn8IHJCuNmiW
Content-Disposition: form-data; name="lastName"

Xavier
------WebKitFormBoundaryqgkaBn8IHJCuNmiW
Content-Disposition: form-data; name="email"

charles@xmen.com
------WebKitFormBoundaryqgkaBn8IHJCuNmiW
Content-Disposition: form-data; name="username"

professorx
------WebKitFormBoundaryqgkaBn8IHJCuNmiW
Content-Disposition: form-data; name="password"

letmein01
------WebKitFormBoundaryqgkaBn8IHJCuNmiW
Content-Disposition: form-data; name="profilePicture"; filename="me.jpg"
Content-Type: image/jpeg

  [[ Binary image data goes here ]]
------WebKitFormBoundaryqgkaBn8IHJCuNmiW--
```

在这个multipart的请求中，我们可以看到`profilePicture`部分与其他部分明显不同。除了其他内容以外，它还有自己的`Content-Type`头，表明它是一个JPEG图片。尽管不一定那么明显，

但profilePicture部分的请求体是二进制数据，而不是简单的文本。

尽管multipart请求看起来很复杂，但在Spring MVC中处理它们却很容易。在编写控制器方法处理文件上传之前，我们必须要配置一个multipart解析器，通过它来告诉DispatcherServlet该如何读取multipart请求。

7.2.1 配置multipart解析器

DispatcherServlet并没有实现任何解析multipart请求数据的功能。它将该任务委托给了Spring中MultipartResolver策略接口的实现，通过这个实现类来解析multipart请求中的内容。从Spring 3.1开始，Spring内置了两个MultipartResolver的实现供我们选择：

- CommonsMultipartResolver：使用Jakarta Commons FileUpload解析multipart请求；
- StandardServletMultipartResolver：依赖于Servlet 3.0对multipart请求的支持（始于Spring 3.1）。

一般来讲，在这两者之间，StandardServletMultipartResolver可能会是优选的方案。它使用Servlet所提供的功能支持，并不需要依赖任何其他的项目。如果我们需要将应用部署到Servlet 3.0之前的容器中，或者还没有使用Spring 3.1或更高版本，那么可能就需要CommonsMultipartResolver了。

使用Servlet 3.0解析multipart请求

兼容Servlet 3.0的StandardServletMultipartResolver没有构造器参数，也没有要设置的属性。这样，在Spring应用上下文中，将其声明为bean就会非常简单，如下所示：

```
@Bean
public MultipartResolver multipartResolver() throws IOException {
  return new StandardServletMultipartResolver();
}
```

既然这个@Bean方法如此简单，你可能就会怀疑我们到底该如何限制StandardServletMultipartResolver的工作方式呢。如果我们想要限制用户上传文件的大小，该怎么实现？如果我们想要指定文件在上传时，临时写入目录在什么位置的话，该如何实现？因为没

有属性和构造器参数，StandardServletMultipartResolver的功能看起来似乎有些受限。

其实并不是这样，我们是有办法配置StandardServletMultipartResolver的限制条件的。只不过不是在Spring中配置StandardServletMultipartResolver，而是要在Servlet中指定multipart的配置。至少，我们必须要指定在文件上传的过程中，所写入的临时文件路径。如果不设定这个最基本配置的话，StandardServlet-MultipartResolver就无法正常工作。具体来讲，我们必须要在web.xml或Servlet初始化类中，将multipart的具体细节作为DispatcherServlet配置的一部分。

如果我们采用Servlet初始化类的方式来配置DispatcherServlet的话，这个初始化类应该已经实现了WebApplicationInitializer，那我们可以在Servlet registration上调用setMultipartConfig()方法，传入一个MultipartConfig-Element实例。如下是最基本的DispatcherServlet multipart配置，它将临时路径设置为“/tmp/spittr/uploads”：

```
DispatcherServlet ds = new DispatcherServlet();
Dynamic registration = context.addServlet("appServlet", ds);
registration.addMapping("/");
registration.setMultipartConfig(
    new MultipartConfigElement("/tmp/spittr/uploads"));
```

如果我们配置DispatcherServlet的Servlet初始化类继承了Abstract AnnotationConfigDispatcherServletInitializer或AbstractDispatcher-ServletInitializer的话，那么我们不会直接创建DispatcherServlet实例并将其注册到Servlet上下文中。这样的话，将不会有对Dynamic Servlet registration的引用供我们使用了。但是，我们可以通过重载customizeRegistration()方法（它会得到一个Dynamic作为参数）来配置multipart的具体细节：

```
@Override
protected void customizeRegistration(Dynamic registration) {
  registration.setMultipartConfig(
      new MultipartConfigElement("/tmp/spittr/uploads"));
}
```

到目前为止，我们所使用是只有一个参数的MultipartConfigElement构造器，这个参数指定的是文件系统中的一个绝对目录，上传文件将会临时写入该目录中。但是，我们还

可以通过其他的构造器来限制上传文件的大小。除了临时路径的位置，其他的构造器所能接受的参数如下：

- 上传文件的最大容量（以字节为单位）。默认是没有限制的。
- 整个multipart请求的最大容量（以字节为单位），不会关心有多少个part以及每个part的大小。默认是没有限制的。
- 在上传的过程中，如果文件大小达到了一个指定最大容量（以字节为单位），将会写入到临时文件路径中。默认值为0，也就是所有上传的文件都会写入到磁盘上。

例如，假设我们想限制文件的大小不超过2MB，整个请求不超过4MB，而且所有的文件都要写到磁盘中。下面的代码使用MultipartConfigElement设置了这些临界值：

```
@Override
protected void customizeRegistration(Dynamic registration) {
  registration.setMultipartConfig(
      new MultipartConfigElement("/tmp/spittr/uploads",
          2097152, 4194304, 0));
}
```

如果我们使用更为传统的web.xml来配置MultipartConfigElement的话，那么可以使用<servlet>中的<multipart-config>元素，如下所示：

```
<servlet>
  <servlet-name>appServlet</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
  <multipart-config>
    <location>/tmp/spittr/uploads</location>
    <max-file-size>2097152</max-file-size>
    <max-request-size>4194304</max-request-size>
  </multipart-config>
</servlet>
```

<multipart-config>的默认值与MultipartConfigElement相同。与MultipartConfigElement一样，必须要配置的是<location>。

配置**Jakarta Commons FileUpload multipart**解析器

通常来讲，StandardServletMultipartResolver会是最佳的选择，但是如果我们需要将应用部署到非Servlet 3.0的容器中，那么就得需要替代的方案。如果喜欢的话，我们可以编写自己的MultipartResolver实现。不过，除非想要在处理multipart请求的

时候执行特定的逻辑，否则的话，没有必要这样做。Spring内置了CommonsMultipartResolver，可以作为StandardServletMultipartResolver的替代方案。

将CommonsMultipartResolver声明为Spring bean的最简单方式如下：

```
@Bean
public MultipartResolver multipartResolver() {
  return new CommonsMultipartResolver();
}
```

与StandardServletMultipartResolver有所不同，CommonsMultipart-Resolver不会强制要求设置临时文件路径。默认情况下，这个路径就是Servlet容器的临时目录。不过，通过设置uploadTempDir属性，我们可以将其指定为一个不同的位置：

```
@Bean
public MultipartResolver multipartResolver() throws IOException {
  CommonsMultipartResolver multipartResolver =
      new CommonsMultipartResolver();
  multipartResolver.setUploadTempDir(
      new FileSystemResource("/tmp/spittr/uploads"));
  return multipartResolver;
}
```

实际上，我们可以按照相同的方式指定其他的multipart上传细节，也就是设置CommonsMultipartResolver的属性。例如，如下的配置就等价于我们在前文通过MultipartConfigElement所配置的StandardServletMultipartResolver：

```
@Bean
public MultipartResolver multipartResolver() throws IOException {
  CommonsMultipartResolver multipartResolver =
          new CommonsMultipartResolver();
  multipartResolver.setUploadTempDir(
      new FileSystemResource("/tmp/spittr/uploads"));
  multipartResolver.setMaxUploadSize(2097152);
  multipartResolver.setMaxInMemorySize(0);
  return multipartResolver;
}
```

在这里，我们将最大的文件容量设置为2MB，最大的内存大小设置为0字节。这两个属性直接对应于MultipartConfigElement的第二个和第四个构造器参数，表明不能上传超过2MB的文件，并且不管文件的大小如何，所有的文件都会写到磁盘中。但是与MultipartConfigElement有所不同，我们无法设定multipart请求整体的最大容量。

7.2.2 处理multipart请求

现在已经在Spring中（或Servlet容器中）配置好了对mutipart请求的处理，那么接下来我们就可以编写控制器方法来接收上传的文件。要实现这一点，最常见的方式就是在某个控制器方法参数上添加`@RequestPart`注解。

假设我们允许用户在注册Spittr应用的时候上传一张图片，那么我们需要修改表单，以允许用户选择要上传的图片，同时还需要修改`SpitterController`中的`processRegistration()`方法来接收上传的图片。如下的代码片段来源于Thymeleaf注册表单视图（registrationForm.html），着重强调了表单所需的修改：

```
<form method="POST" th:object="${spitter}"
      enctype="multipart/form-data">
...
    <label>Profile Picture</label>:
      <input type="file"
             name="profilePicture"
             accept="image/jpeg,image/png,image/gif" /><br/>
...
</form>
```

`<form>`标签现在将`enctype`属性设置为`multipart/form-data`，这会告诉浏览器以multipart数据的形式提交表单，而不是以表单数据的形式进行提交。在multipart中，每个输入域都会对应一个part。

除了注册表单中已有的输入域，我们还添加了一个新的`<input>`域，其`type`为`file`。这能够让用户选择要上传的图片文件。accept属性用来将文件类型限制为JPEG、PNG以及GIF图片。根据其`name`属性，图片数据将会发送到multipart请求中的`profilePicture` part之中。

现在，我们需要修改`processRegistration()`方法，使其能够接受上传的图片。其中一种方式是添加byte数组参数，并为其添加`@RequestPart`注解。如下为示例：

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    @RequestPart("profilePicture") byte[] profilePicture,
    @Valid Spitter spitter,
    Errors errors) {
  ...
}
```

当注册表单提交的时候，profilePicture属性将会给定一个byte数组，这个数组中包含了请求中对应part的数据（通过@RequestPart指定）。如果用户提交表单的时候没有选择文件，那么这个数组会是空（而不是null）。获取到图片数据后，processRegistration()方法剩下的任务就是将文件保存到某个位置。

我们将会稍后讨论如何保存文件。但首先，想一下，对于提交的图片数据我们都了解哪些信息呢。或者，更为重要的是，我们还不知道些什么呢？尽管我们已经得到了byte数组形式的图片数据，并且根据它能够得到图片的大小，但是对于其他内容我们就一无所知了。我们不知道文件的类型是什么，甚至不知道原始的文件名是什么。你需要判断如何将byte数组转换为可存储的文件。

接受**MultipartFile**

使用上传文件的原始byte比较简单但是功能有限。因此，Spring还提供了MultipartFile接口，它为处理multipart数据提供了内容更为丰富的对象。如下的程序清单展现了MultipartFile接口的概况。

程序清单**7.5**　**Spring**所提供的**MultipartFile**接口，用来处理上传的文件

```
package org.springframework.web.multipart;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;

public interface MultipartFile {
  String getName();
  String getOriginalFilename();
  String getContentType();
  boolean isEmpty();
  long getSize();
  byte[] getBytes() throws IOException;
  InputStream getInputStream() throws IOException;
  void transferTo(File dest) throws IOException;
}
```

我们可以看到，MultipartFile提供了获取上传文件byte的方式，但是它所提供的功能并不仅限于此，还能获得原始的文件名、大小以及内容类型。它还提供了一个InputStream，用来将文件数据以流的方式进行读取。

除此之外，MultipartFile还提供了一个便利的transferTo()方法，它能够帮助我们将上传的文件写入到文件系统中。作为样例，我们可以在process-Registration()方法中添加如下的几行代

码，从而将上传的图片文件写入到文件系统中：

```
profilePicture.transferTo(
    new File("/data/spittr/" + profilePicture.getOriginalFilename()));
```

将文件保存到本地文件系统中是非常简单的，但是这需要我们对这些文件进行管理。我们需要确保有足够的空间，确保当出现硬件故障时，文件进行了备份，还需要在集群的多个服务器之间处理这些图片文件的同步。

将文件保存到**Amazon S3**中

另外一种方案就是让别人来负责处理这些事情。多加几行代码，我们就能将图片保存到云端。例如，如下的程序清单所展现的`saveImage()`方法能够将上传的文件保存到Amazon S3中，我们在`processRegistration()`中可以调用该方法。

程序清单**7.6**　将**MultipartFile**保存到**Amazon S3**中

```
private void saveImage(MultipartFile image)
    throws ImageUploadException {
    try {
      AWSCredentials awsCredentials =
          new AWSCredentials(s3AccessKey, s2SecretKey);
      S3Service s3 = new RestS3Service(awsCredentials);      <- 构建 S3 服务

      S3Bucket bucket = s3.getBucket("spittrImages");        <- 创建 S3 bucket 和 object
      S3Object imageObject =
          new S3Object(image.getOriginalFilename());

      imageObject.setDataInputStream(                        <- 设置图片数据
          image.getInputStream());
      imageObject.setContentLength(image.getSize());
      imageObject.setContentType(image.getContentType());

      AccessControlList acl = new AccessControlList();      <- 设置权限
      acl.setOwner(bucket.getOwner());
      acl.grantPermission(GroupGrantee.ALL_USERS,
          Permission.PERMISSION_READ);
      imageObject.setAcl(acl);

      s3.putObject(bucket, imageObject);                     <- 保存图片
    } catch (Exception e) {
      throw new ImageUploadException("Unable to save image", e);
    }
  }
```

`saveImage()`方法所做的第一件事就是构建Amazon Web Service（AWS）凭证。为了完成这一点，你需要有一个S3 Access Key和S3 Secret Access Key。当注册S3服务的时候，Amazon会将其提供给你。它们会通过值注入的方式提供给`Spitter-Controller`。

AWS凭证准备好后，`saveImage()`方法创建了一个JetS3t的`RestS3Service`实例，可以通过它来操作S3文件系统。它获取

spitterImages bucket的引用并创建用来包含图片的S3Object对象，接下来将图片数据填充到S3Object。

在调用putObject()方法将图片数据写到S3之前，saveImage()方法设置了S3Object的权限，从而允许所有的用户查看它。这是很重要的——如果没有它的话，这些图片对我们应用程序的用户就是不可见的。最后，如果出现任何问题的话，将会抛出ImageUploadException异常。

以**Part**的形式接受上传的文件

如果你需要将应用部署到Servlet 3.0的容器中，那么会有MultipartFile的一个替代方案。Spring MVC也能接受javax.servlet.http.Part作为控制器方法的参数。如果使用Part来替换MultipartFile的话，那么processRegistration()的方法签名将会变成如下的形式：

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    @RequestPart("profilePicture") Part profilePicture,
    @Valid Spitter spitter,
    Errors errors) {
  ...
}
```

就主体来言（不开玩笑地说），Part接口与MultipartFile并没有太大的差别。在如下的程序清单中，我们可以看到Part接口的有一些方法其实是与MultipartFile相对应的。

程序清单**7.7**　**Part**接口：**Spring MultipartFile**的替代方案

```
package javax.servlet.http;
import java.io.*;
import java.util.*;

public interface Part {
  public InputStream getInputStream() throws IOException;
  public String getContentType();
  public String getName();
  public String getSubmittedFileName();
  public long getSize();
  public void write(String fileName) throws IOException;
  public void delete() throws IOException;
  public String getHeader(String name);
  public Collection<String> getHeaders(String name);
  public Collection<String> getHeaderNames();
}
```

在很多情况下，Part方法的名称与MultipartFile方法的名称是完全相同的。有一些比较类似，但是稍有差异，比如

getSubmittedFileName()对应于getOriginalFilename()。类似地，write()对应于transferTo()，借助该方法我们能够将上传的文件写入文件系统中：

```
profilePicture.write("/data/spittr/" +
        profilePicture.getOriginalFilename());
```

值得一提的是，如果在编写控制器方法的时候，通过Part参数的形式接受文件上传，那么就没有必要配置MultipartResolver了。只有使用MultipartFile的时候，我们才需要MultipartResolver。

7.3 处理异常

到现在为止，在Spittr应用中，我们假设所有的功能都正常运行。但是如果某个地方出错的话，该怎么办呢？当处理请求的时候，抛出异常该怎么处理呢？如果发生了这样的情况，该给客户端什么响应呢？

不管发生什么事情，不管是好的还是坏的，Servlet请求的输出都是一个Servlet响应。如果在请求处理的时候，出现了异常，那它的输出依然会是Servlet响应。异常必须要以某种方式转换为响应。

Spring提供了多种方式将异常转换为响应：

- 特定的Spring异常将会自动映射为指定的HTTP状态码；
- 异常上可以添加@ResponseStatus注解，从而将其映射为某一个HTTP状态码；
- 在方法上可以添加@ExceptionHandler注解，使其用来处理异常。

处理异常的最简单方式就是将其映射到HTTP状态码上，进而放到响应之中。接下来，我们看一下如何将异常映射为某一个HTTP状态码。

7.3.1 将异常映射为HTTP状态码

在默认情况下，Spring会将自身的一些异常自动转换为合适的状态码。表7.1列出了这些映射关系。

表**7.1** **Spring**的一些异常会默认映射为**HTTP**状态码

Spring异常	HTTP状态码
BindException	400 - Bad Request
ConversionNotSupportedException	500 - Internal Server Error
HttpMediaTypeNotAcceptableException	406 - Not Acceptable
HttpMediaTypeNotSupportedException	415 - Unsupported Media Type
HttpMessageNotReadableException	400 - Bad Request
HttpMessageNotWritableException	500 - Internal Server Error
HttpRequestMethodNotSupportedException	405 - Method Not Allowed
MethodArgumentNotValidException	400 - Bad Request
MissingServletRequestParameterException	400 - Bad Request
MissingServletRequestPartException	400 - Bad Request
NoSuchRequestHandlingMethodException	404 - Not Found
TypeMismatchException	400 - Bad Request

表7.1中的异常一般会由Spring自身抛出，作为DispatcherServlet处理过程中或执行校验时出现问题的结果。例如，如果DispatcherServlet无法找到适合处理请求的控制器方法，那么将会抛出NoSuchRequestHandlingMethodException异常，最终的结果就是产生404状态码的响应（Not Found）。

尽管这些内置的映射是很有用的，但是对于应用所抛出的异常它们就无能为力了。幸好，Spring提供了一种机制，能够通过@ResponseStatus注解将异常映射为HTTP状态码。

为了阐述这项功能，请参考SpittleController中如下的请求处理方法，它可能会产生HTTP 404状态（但目前还没有实现）：

```
@RequestMapping(value="/{spittleId}", method=RequestMethod.GET)
public String spittle(
    @PathVariable("spittleId") long spittleId,
    Model model) {
  Spittle spittle = spittleRepository.findOne(spittleId);
  if (spittle == null) {
    throw new SpittleNotFoundException();
  }
  model.addAttribute(spittle);
  return "spittle";
}
```

在这里，会从SpittleRepository中，通过ID检索Spittle对象。如果findOne()方法能够返回Spittle对象的话，那么会将Spittle放到模型中，然后名为spittle的视图会负责将其渲染到响应之中。但是如果findOne()方法返回null的话，那么将会抛出SpittleNotFoundException异常。现在SpittleNotFoundException就是一个简单的非检查型异常，如下所示：

```
package spittr.web;
public class SpittleNotFoundException extends RuntimeException {
}
```

如果调用spittle()方法来处理请求，并且给定ID获取到的结果为空，那么SpittleNotFoundException（默认）将会产生500状态码（Internal Server Error）的响应。实际上，如果出现任何没有映射的异常，响应都会带有500状态码，但是，我们可以通过映射SpittleNotFoundException对这种默认行为进行变更。

当抛出SpittleNotFoundException异常时，这是一种请求资源没有找到的场景。如果资源没有找到的话，HTTP状态码404是最为精确的响应状态码。所以，我们要使用@ResponseStatus注解将SpittleNotFoundException映射为HTTP状态码404。

程序清单**7.8　@ResponseStatus**注解：将异常映射为特定的状态码

```
package spittr.web;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value=HttpStatus.NOT_FOUND,          <-- 将异常映射为HTTP 状态404
                reason="Spittle Not Found")
public class SpittleNotFoundException extends RuntimeException {
}
```

在引入@ResponseStatus注解之后，如果控制器方法抛出SpittleNotFound-Exception异常的话，响应将会具有404状态码，这是因为Spittle Not Found。

7.3.2 编写异常处理的方法

在很多的场景下，将异常映射为状态码是很简单的方案，并且就功能来说也足够了。但是如果我们想在响应中不仅要包括状态码，还要包含所产生的错误，那该怎么办呢？此时的话，我们就不能将异常视为HTTP错误了，而是要按照处理请求的方式来处理异常了。

作为样例，假设用户试图创建的Spittle与已创建的Spittle文本完全相同，那么SpittleRepository的save()方法将会抛出DuplicateSpittle Exception异常。这意味着SpittleController的saveSpittle()方法可能需要处理这个异常。如下面的程序清单所示，saveSpittle()方法可以直接处理这个异常。

程序清单**7.9** 在处理请求的方法中直接处理异常

```
@RequestMapping(method=RequestMethod.POST)
public String saveSpittle(SpittleForm form, Model model) {
  try {
    spittleRepository.save(
        new Spittle(null, form.getMessage(), new Date(),
            form.getLongitude(), form.getLatitude()));
    return "redirect:/spittles";
  } catch (DuplicateSpittleException e) {          <-- 捕获异常
     return "error/duplicate";
  }
}
```

程序清单7.9中并没有特别之处，它只是在Java中处理异常的基本样例，除此之外，也就没什么了。

它运行起来没什么问题，但是这个方法有些复杂。该方法可以有两个路径，每个路径会有不同的输出。如果能让saveSpittle()方法只关注正确的路径，而让其他方法处理异常的话，那么它就能简单一些。

首先，让我们首先将saveSpittle()方法中的异常处理方法剥离掉：

```
@RequestMapping(method=RequestMethod.POST)
public String saveSpittle(SpittleForm form, Model model) {
  spittleRepository.save(
      new Spittle(null, form.getMessage(), new Date(),
          form.getLongitude(), form.getLatitude()));
  return "redirect:/spittles";
}
```

可以看到，saveSpittle()方法简单了许多。因为它只关注成功保存Spittle的情况，所以只有一个执行路径，很容易理解（和测试）。

现在，我们为SpittleController添加一个新的方法，它会处理抛出DuplicateSpittleException的情况：

```
@ExceptionHandler(DuplicateSpittleException.class)
public String handleDuplicateSpittle() {
  return "error/duplicate";
}
```

handleDuplicateSpittle()方法上添加了@ExceptionHandler注解，当抛出DuplicateSpittleException异常的时候，将会委托该方法来处理。它返回的是一个String，这与处理请求的方法是一致的，指定了要渲染的逻辑视图名，它能够告诉用户他们正在试图创建一条重复的条目。

对于@ExceptionHandler注解标注的方法来说，比较有意思的一点在于它能处理同一个控制器中所有处理器方法所抛出的异常。所以，尽管我们从saveSpittle()中抽取代码创建了handleDuplicateSpittle()方法，但是它能够处理SpittleController中所有方法所抛出的DuplicateSpittleException异常。我们不用在每一个可能抛出DuplicateSpittleException的方法中添加异常处理代码，这一个方法就涵盖了所有的功能。

既然@ExceptionHandler注解所标注的方法能够处理同一个控制器类中所有处理器方法的异常，那么你可能会问有没有一种方法能够处理所有控制器中处理器方法所抛出的异常呢。从Spring 3.2开始，这肯定是能够实现的，我们只需将其定义到控制器通知类中即可。

什么是控制器通知方法？很高兴你会问这样的问题，因为这就是我们

下面要讲的内容。

7.4　为控制器添加通知

如果控制器类的特定切面能够运用到整个应用程序的所有控制器中，那么这将会便利很多。举例来说，如果要在多个控制器中处理异常，那@ExceptionHandler注解所标注的方法是很有用的。不过，如果多个控制器类中都会抛出某个特定的异常，那么你可能会发现要在所有的控制器方法中重复相同的@ExceptionHandler方法。或者，为了避免重复，我们会创建一个基础的控制器类，所有控制器类要扩展这个类，从而继承通用的@ExceptionHandler方法。

Spring 3.2为这类问题引入了一个新的解决方案：控制器通知。控制器通知（controller advice）是任意带有@ControllerAdvice注解的类，这个类会包含一个或多个如下类型的方法：

- @ExceptionHandler注解标注的方法；
- @InitBinder注解标注的方法；
- @ModelAttribute注解标注的方法。

在带有@ControllerAdvice注解的类中，以上所述的这些方法会运用到整个应用程序所有控制器中带有@RequestMapping注解的方法上。

@ControllerAdvice注解本身已经使用了@Component，因此@ControllerAdvice注解所标注的类将会自动被组件扫描获取到，就像带有@Component注解的类一样。

@ControllerAdvice最为实用的一个场景就是将所有的@ExceptionHandler方法收集到一个类中，这样所有控制器的异常就能在一个地方进行一致的处理。例如，我们想将DuplicateSpittleException的处理方法用到整个应用程序的所有控制器上。如下的程序清单展现的AppWideExceptionHandler就能完成这一任务，这是一个带有@ControllerAdvice注解的类。

程序清单**7.10**　使用**@ControllerAdvice**，为所有的控制器处理异常

```
package spitter.web;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice                                          <-- 定义控制器类
public class AppWideExceptionHandler {

  @ExceptionHandler(DuplicateSpittleException.class)       <-- 定义异常处理方法
  public String duplicateSpittleHandler() {
    return "error/duplicate";
  }

}
```

现在，如果任意的控制器方法抛出了DuplicateSpittleException，不管这个方法位于哪个控制器中，都会调用这个duplicateSpittleHandler()方法来处理异常。我们可以像编写@RequestMapping注解的方法那样来编写@ExceptionHandler注解的方法。如程序清单7.10所示，它返回“error/duplicate”作为逻辑视图名，因此将会为用户展现一个友好的出错页面。

7.5 跨重定向请求传递数据

在5.4.1小节中，在处理完POST请求后，通常来讲一个最佳实践就是执行一下重定向。除了其他的一些因素外，这样做能够防止用户点击浏览器的刷新按钮或后退箭头时，客户端重新执行危险的POST请求。

在第5章，在控制器方法返回的视图名称中，我们借助了“redirect:”前缀的力量。当控制器方法返回的String值以“redirect:”开头的话，那么这个String不是用来查找视图的，而是用来指导浏览器进行重定向的路径。我们可以回头看一下程序清单5.17，可以看到processRegistration()方法返回的“redirect:String”如下所示：

```
return "redirect:/spitter/" + spitter.getUsername();
```

“redirect:”前缀能够让重定向功能变得非常简单。你可能会想Spring很难再让重定向功能变得更简单了。但是，请稍等：Spring为重定向功能还提供了一些其他的辅助功能。

具体来讲，正在发起重定向功能的方法该如何发送数据给重定向的目标方法呢？一般来讲，当一个处理器方法完成之后，该方法所指定的模型数据将会复制到请求中，并作为请求中的属性，请求会转发（forward）到视图上进行渲染。因为控制器方法和视图所处理的是

同一个请求，所以在转发的过程中，请求属性能够得以保存。

但是，如图7.1所示，当控制器的结果是重定向的话，原始的请求就结束了，并且会发起一个新的`GET`请求。原始请求中所带有的模型数据也就随着请求一起消亡了。在新的请求属性中，没有任何的模型数据，这个请求必须要自己计算数据。

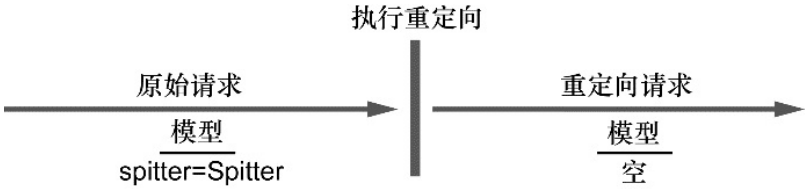

图7.1　模型的属性是以请求属性的形式存放在请求中的，在重定向后无法存活

显然，对于重定向来说，模型并不能用来传递数据。但是我们也有一些其他方案，能够从发起重定向的方法传递数据给处理重定向方法中：

- 使用URL模板以路径变量和/或查询参数的形式传递数据；
- 通过flash属性发送数据。

首先，我们看一下Spring如何帮助我们通过路径变量和/或查询参数的形式传递数据。

7.5.1　通过URL模板进行重定向

通过路径变量和查询参数传递数据看起来非常简单。例如，在程序清单5.19中，我们以路径变量的形式传递了新创建`Spitter`的`username`。但是按照现在的写法，`username`的值是直接连接到重定向`String`上的。这能够正常运行，但是还远远不能说没有问题。当构建URL或SQL查询语句的时候，使用`String`连接是很危险的。

```
return "redirect:/spitter/{username}";
```

除了连接String的方式来构建重定向URL，Spring还提供了使用模板的方式来定义重定向URL。例如，在程序清单5.19中，`processRegistration()`方法的最后一行可以改写为如下的形式：

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    Spitter spitter, Model model) {
  spitterRepository.save(spitter);

  model.addAttribute("username", spitter.getUsername());
  return "redirect:/spitter/{username}";
}
```

现在，username作为占位符填充到了URL模板中，而不是直接连接到重定向String中，所以username中所有的不安全字符都会进行转义。这样会更加安全，这里允许用户输入任何想要的内容作为username，并会将其附加到路径上。

除此之外，模型中所有其他的原始类型值都可以添加到URL中作为查询参数。作为样例，假设除了username以外，模型中还要包含新创建Spitter对象的id属性，那processRegistration()方法可以改写为如下的形式：

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    Spitter spitter, Model model) {
  spitterRepository.save(spitter);
  model.addAttribute("username", spitter.getUsername());
  model.addAttribute("spitterId", spitter.getId());
  return "redirect:/spitter/{username}";
}
```

所返回的重定向String并没有太大的变化。但是，因为模型中的spitterId属性没有匹配重定向URL中的任何占位符，所以它会自动以查询参数的形式附加到重定向URL上。

如果username属性的值是habuma并且spitterId属性的值是42，那么结果得到的重定向URL路径将会是“/spitter/habuma?spitterId=42”。

通过路径变量和查询参数的形式跨重定向传递数据是很简单直接的方式，但它也有一定的限制。它只能用来发送简单的值，如String和数字的值。在URL中，并没有办法发送更为复杂的值，但这正是flash属性能够提供帮助的领域。

7.5.2 使用flash属性

假设我们不想在重定向中发送username或ID了，而是要发送实际的Spitter对象。如果我们只发送ID的话，那么处理重定向的方法还需要从数据库中查找才能得到Spitter对象。但是，在重定向之前，我们其实已经得到了Spitter对象。为什么不将其发送给处理

重定向的方法，并将其展现出来呢？

Spitter对象要比String和int更为复杂。因此，我们不能像路径变量或查询参数那么容易地发送Spitter对象。它只能设置为模型中的属性。

但是，正如我们前面所讨论的那样，模型数据最终是以请求参数的形式复制到请求中的，当重定向发生的时候，这些数据就会丢失。因此，我们需要将Spitter对象放到一个位置，使其能够在重定向的过程中存活下来。

有个方案是将Spitter放到会话中。会话能够长期存在，并且能够跨多个请求。所以我们可以在重定向发生之前将Spitter放到会话中，并在重定向后，从会话中将其取出。当然，我们还要负责在重定向后在会话中将其清理掉。

实际上，Spring也认为将跨重定向存活的数据放到会话中是一个很不错的方式。但是，Spring认为我们并不需要管理这些数据，相反，Spring提供了将数据发送为flash属性（flash attribute）的功能。按照定义，flash属性会一直携带这些数据直到下一次请求，然后才会消失。

Spring提供了通过RedirectAttributes设置flash属性的方法，这是Spring 3.1引入的Model的一个子接口。RedirectAttributes提供了Model的所有功能，除此之外，还有几个方法是用来设置flash属性的。

具体来讲，RedirectAttributes提供了一组addFlashAttribute()方法来添加flash属性。重新看一下processRegistration()方法，我们可以使用addFlashAttribute()将Spitter对象添加到模型中：

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    Spitter spitter, RedirectAttributes model) {
  spitterRepository.save(spitter);
  model.addAttribute("username", spitter.getUsername());
  model.addFlashAttribute("spitter", spitter);
  return "redirect:/spitter/{username}";
}
```

在这里，我们调用了addFlashAttribute()方法，并将spitter作为key，Spitter对象作为值。另外，我们还可以不设置key参数，让key根据值的类型自行推断得出：

```
model.addFlashAttribute(spitter);
```

因为我们传递了一个Spitter对象给addFlashAttribute()方法，所以推断得到的key将会是spitter。

在重定向执行之前，所有的flash属性都会复制到会话中。在重定向后，存在会话中的flash属性会被取出，并从会话转移到模型之中。处理重定向的方法就能从模型中访问Spitter对象了，就像获取其他的模型对象一样。图7.2阐述了它是如何运行的。

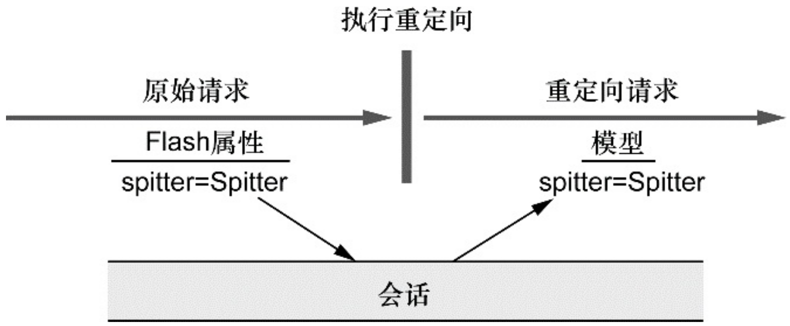

图7.2　flash属性保存在会话中，然后再放到模型中，因此能够在重定向的过程中存活

为了完成flash属性的流程，如下展现了更新版本的showSpitterProfile()方法，在从数据库中查找之前，它会首先从模型中检查Spitter对象：

```
@RequestMapping(value="/{username}", method=GET)
public String showSpitterProfile(
        @PathVariable String username, Model model) {
  if (!model.containsAttribute("spitter")) {
    model.addAttribute(
        spitterRepository.findByUsername(username));
  }
  return "profile";
}
```

可以看到，showSpitterProfile()方法所做的第一件事就是检查是否存有key为spitter的model属性。如果模型中包含spitter属性，那就什么都不用做了。这里面包含的Spitter对象将会传递到视图中进行渲染。但是如果模型中不包含spitter属性的话，那么showSpitterProfile()将会从Repository中查找Spitter，并将其存放到模型中。

7.6　小结

在Spring中，总是会有“还没有结束”的感觉：更多的特性、更多的选择以及实现开发目标的更多方式。Spring MVC有很多功能和技巧。

当然，Spring MVC的环境搭建是有多种可选方案的一个领域。在本章中，我们首先看了一下搭建Spring MVC中`DispatcherServlet`和`ContextLoaderListener`的多种方式。我们还看到了如何调整`DispatcherServlet`的注册功能以及如何注册自定义的Servlet和Filter。如果你需要将应用部署到更老的应用服务器上，我们还快速了解了如何使用web.xml声明`DispatcherServlet`和`ContextLoaderListener`。

然后，我们了解了如何处理Spring MVC控制器所抛出的异常。尽管带有`@RequestMapping`注解的方法可以在自身的代码中处理异常，但是如果我们将异常处理的代码抽取到单独的方法中，那么控制器的代码会整洁得多。

为了采用一致的方式处理通用的任务，包括在应用的所有控制器中处理异常，Spring 3.2引入了`@ControllerAdvice`，它所创建的类能够将控制器的通用行为抽取到同一个地方。

最后，我们看了一下如何跨重定向传递数据，包括Spring对flash属性的支持：类似于模型的属性，但是能在重定向后存活下来。这样的话，就能采用非常恰当的方式为`POST`请求执行一个重定向回应，而且能够将处理`POST`请求时的模型数据传递过来，然后在重定向后使用或展现这些模型数据。

如果你还有疑惑的话，那么可以告诉你，这就是我所说的“更多的功能”！其实，我们并没有讨论到Spring MVC的每个方面。我们将会在第16章中重新讨论Spring MVC，到时你会看到如何使用它来创建REST API。

但现在，我们将会暂时放下Spring MVC，看一下Spring Web Flow，这是一个构建在Spring MVC之上的流程框架，它能够引导用户执行一系列向导步骤。

第8章　使用Spring Web Flow

本章内容：

- 创建会话式的Web应用程序
- 定义流程状态和行为
- 保护Web流程

关于互联网，很奇妙的一件事就是它很容易让你迷失。有如此之多的内容可以查看和阅读，而超链接是互联网强大魔力的核心。无怪乎将其称为网，正如蜘蛛织出的网，它会将经过的任何东西困住。我必须承认：之所以在编写此书时花费了如此多的时间，其中的一个原因就是我曾经迷失在维基百科无休无止的链接之中。

有时候，Web应用程序需要控制网络冲浪者的方向，引导他们一步步地访问应用。比较典型的例子就是电子商务站点的结账流程，从购物车开始，应用程序会引导你依次经过派送详情、账单信息以及最终的订单确认流程。

Spring Web Flow是一个Web框架，它适用于元素按规定流程运行的程序。在本章中，我们将会探索Spring Web Flow并了解它如何应用于Spring Web框架平台。

其实我们可以使用任何Web框架编写流程化的应用程序。我曾经看到过一个应用程序，在Struts中构建了特定的流程。但是这样就没有办法将流程与实现分开了，你会发现流程的定义分散在组成流程的各个元素中。没有地方能够完整地描述整个流程。

Spring Web Flow是Spring MVC的扩展，它支持开发基于流程的应用程序。它将流程的定义与实现流程行为的类和视图分离开来。

在介绍Spring Web Flow的时候，我们将暂时放下Spittr样例并使用生成披萨订单的新Web应用程序。我们会使用Spring Web Flow来定义订单流程。

使用Spring Web Flow的第一步是在项目中安装它。让我们从这里开始吧。

8.1 在Spring中配置Web Flow

Spring Web Flow是构建于Spring MVC基础之上的。这意味着所有的流程请求都需要首先经过Spring MVC的`DispatcherServlet`。我们需要在Spring应用上下文中配置一些bean来处理流程请求并执行流程。

现在，还不支持在Java中配置Spring Web Flow，所以我们别无选择，只能在XML中对其进行配置。有一些bean会使用Spring Web Flow的Spring配置文件命名空间来进行声明。因此，我们需要在上下文定义XML文件中添加这个命名空间声明：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:flow="http://www.springframework.org/schema/webflow-config"
 xsi:schemaLocation=
  "http://www.springframework.org/schema/webflow-config
   http://www.springframework.org/schema/webflow-config/[CA]
   spring-webflow-config-2.3.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
```

在声明了命名空间之后，我们就为装配Web Flow的bean做好了准备，让我们从流程执行器（flow executor）开始吧。

8.1.1 装配流程执行器

正如其名字所示，流程执行器（flow executor）驱动流程的执行。当用户进入一个流程时，流程执行器会为用户创建并启动一个流程执行实例。当流程暂停的时候（如为用户展示视图时），流程执行器会在用户执行操作后恢复流程。

在Spring中，`<flow:flow-executor>`元素会创建一个流程执行器：

```
<flow:flow-executor id="flowExecutor" />
```

尽管流程执行器负责创建和执行流程，但它并不负责加载流程定义。这个责任落在了流程注册表（flow registry）身上，接下来我们会创建它。

8.1.2 配置流程注册表

流程注册表（flow registry）的工作是加载流程定义并让流程执行器能够使用它们。我们可以在Spring中使用<flow:flow-registry>配置流程注册表，如下所示：

```
<flow:flow-registry id="flowRegistry" base-path="/WEB-INF/flows">
   <flow:flow-location-pattern value="*-flow.xml" />
</flow:flow-registry>
```

在这里的声明中，流程注册表会在“/WEB-INF/flows”目录下查找流程定义，这是通过base-path属性指明的。依据<flow:flow-location-pattern>元素的值，任何文件名以“-flow.xml”结尾的XML文件都将视为流程定义。

所有的流程都是通过其ID来进行引用的。这里我们使用了<flow:flow- ocation-pattern>元素，流程的ID就是相对于base-path的路径——或者双星号所代表的路径。图8.1展示了示例中的流程ID是如何计算的。

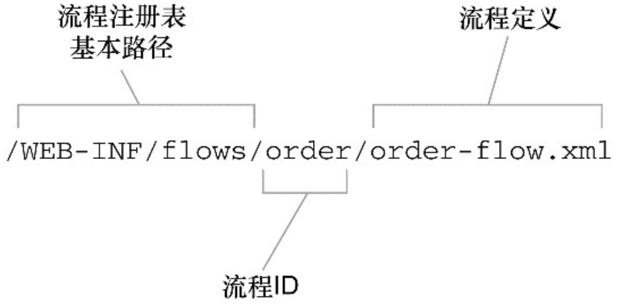

图8.1　在使用流程定位模式的时候，
流程定义文件相对于基本路径的路径将被用作流程的ID

作为另一种方式，我们可以去除base-path属性，而显式声明流程定义文件的位置：

```
<flow:flow-registry id="flowRegistry">
   <flow:flow-location path="/WEB-INF/flows/springpizza.xml" />
</flow:flow-registry>
```

在这里，使用了<flow:flow-location>而不是<flow:flow-location-pattern>，path属性直接指明了“/WEB-INF/flows/springpizza.xml”作为流程定义。当我们这样配置的话，流程的ID是从流程定义文件的文件名中获得的，在这里就是springpizza。

如果你希望更显式地指定流程ID，那你可以通过<flow:flow-location>元素的id属性来进行设置。例如，要将pizza作为流程ID，可以像这样配置：

```
<flow:flow-registry id="flowRegistry">
   <flow:flow-location id="pizza"
         path="/WEB-INF/flows/springpizza.xml" />
</flow:flow-registry>
```

8.1.3 处理流程请求

我们在前一章曾经看到，DispatcherServlet一般将请求分发给控制器。但是对于流程而言，我们需要一个FlowHandlerMapping来帮助DispatcherServlet将流程请求发送给Spring Web Flow。在Spring应用上下文中，FlowHandlerMapping的配置如下：

```
<bean class=
        "org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
  <property name="flowRegistry" ref="flowRegistry" />
</bean>
```

你可以看到，FlowHandlerMapping装配了流程注册表的引用，这样它就能知道如何将请求的URL匹配到流程上。例如，如果我们有一个ID为pizza的流程，FlowHandlerMapping就会知道如果请求的URL模式（相对于应用程序的上下文路径）是“/pizza”的话，就要将其匹配到这个流程上。

然而，FlowHandlerMapping的工作仅仅是将流程请求定向到Spring Web Flow上，响应请求的是FlowHandlerAdapter。FlowHandlerAdapter等同于Spring MVC的控制器，它会响应发送的流程请求并对其进行处理。FlowHandlerAdapter可以像下面这样装配成一个Spring bean，如下所示：

```
<bean class=
       "org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
  <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

这个处理适配器是DispatcherServlet和Spring Web Flow之间的桥梁。它会处理流程请求并管理基于这些请求的流程。在这里，它装配了流程执行器的引用，而后者是为所处理的请求执行流程的。

我们已经配置了Spring WebFlow所需的bean和组件。剩下就是真正定义流程了。我们随后将会进行这项工作。但首先，让我们先了解一下组成流程的元素。

8.2 流程的组件

在Spring Web Flow中，流程是由三个主要元素定义的：状态、转移和流程数据。状态（State）是流程中事件发生的地点。如果你将流程想象成公路旅行，那状态就是路途上的城镇、路边饭店以及风景点。流程中的状态是业务逻辑执行、做出决策或将页面展现给用户的地方，而不是在公路旅行中买Doritos薯片和健怡可乐的所在。

如果流程状态就像公路旅行中停下来的地点，那转移（transition）就是连接这些点的公路。在流程中，你通过转移的方式从一个状态到另一个状态。

当你在城镇之间旅行的时候，你可能要买一些纪念品，留下一些记忆并在路上取一些空的零食袋。类似地，在流程处理中，它要收集一些数据：流程的当前状况。我很想将其称为流程的状态，但是在我们讨论流程的时候状态（state）已经有了另外的含义。

让我们仔细看一下在Spring Web Flow中这三个元素是如何定义的。

8.2.1 状态

Spring Web Flow定义了五种不同类型的状态，如表8.1所示。通过选择Spring Web Flow的状态几乎可以把任意的安排功能构造成会话式的Web应用。尽管并不是所有的流程都需要表8.1所描述的状态，但最终你可能会经常使用它们中的大多数。

表**8.1 Spring Web Flow**可供选择的状态

状 态 类 型	它是用来做什么的
行为（Action）	行为状态是流程逻辑发生的地方
决策（Decision）	决策状态将流程分成两个方向，它会基于流程数据的评估结果确定流程方向
结束（End）	结束状态是流程的最后一站。一旦进入End状态，流程就会终止
子流程（Subflow）	子流程状态会在当前正在运行的流程上下文中启动一个新的流程

视图（View）	视图状态会暂停流程并邀请用户参与流程

稍后我们将会看到如何将这些不同类型的状态组合起来形成一个完整的流程。但首先，让我们了解一下这些流程元素在Spring Web Flow定义中是如何表现的。

视图状态

视图状态用于为用户展现信息并使用户在流程中发挥作用。实际的视图实现可以是Spring支持的任意视图类型，但通常是用JSP来实现的。

在流程定义的XML文件中，`<view-state>`用于定义视图状态：

```
<view-state id="welcome" />
```

在这个简单的示例中，`id`属性有两个含义。它在流程内标示这个状态。除此以外，因为在这里没有在其他地方指定视图，所以它也指定了流程到达这个状态时要展现的逻辑视图名为`welcome`。

如果你愿意显式指定另外一个视图名，那可以使用`view`属性做到这一点：

```
<view-state id="welcome" view="greeting" />
```

如果流程为用户展现了一个表单，你可能希望指明表单所绑定的对象。为了做到这一点，可以设置`model`属性：

```
<view-state id="takePayment" model="flowScope.paymentDetails"/>
```

这里我们指定`takePayment`视图中的表单将绑定流程作用域内的`paymentDetails`对象。（稍后，我们将会更详细地介绍流程作用域和数据。）

行为状态

视图状态会涉及到流程应用程序的用户，而行为状态则是应用程序自身在执行任务。行为状态一般会触发Spring所管理bean的一些方法并根据方法调用的执行结果转移到另一个状态。

在流程定义XML中，行为状态使用`<action-state>`元素来声明。

这里是一个例子：

```
<action-state id="saveOrder">
  <evaluate expression="pizzaFlowActions.saveOrder(order)" />
  <transition to="thankYou" />
</action-state>
```

尽管不是严格需要的，但是<action-state>元素一般都会有一个<evaluate>作为子元素。<evaluate>元素给出了行为状态要做的事情。expression属性指定了进入这个状态时要评估的表达式。在本示例中，给出的expression是SpEL表达式，它表明将会找到ID为pizzaFlowActions的bean并调用其saveOrder()方法。

Spring Web Flow与表达式语言

在这几年以来，Spring Web Flow在选择的表达式语言方面，经过了一些变化。在1.0版本的时候，Spring Web Flow使用的是对象图导航语言（Object-Graph Navigation Language，OGNL）。随后的2.0版本又换成了统一表达式语言（Unified Expression Language，Unified EL）。在2.1版本中，Spring Web Flow使用的是SpEL。

尽管可以使用上述的任意表达式语言来配置Spring Web Flow，但SpEL是默认和推荐使用的表达式语言。因此，当定义流程的时候，我们会选择使用SpEL，忽略掉其他的可选方案。

决策状态

有可能流程会完全按照线性执行，从一个状态进入另一个状态，没有其他的替代路线。但是更常见的情况是流程在某一个点根据流程的当前情况进入不同的分支。

决策状态能够在流程执行时产生两个分支。决策状态将评估一个Boolean类型的表达式，然后在两个状态转移中选择一个，这要取决于表达式会计算出true还是false。在XML流程定义中，决策状态通过<decision-state>元素进行定义。典型的决策状态示例如下所示：

```
<decision-state id="checkDeliveryArea">
  <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
      then="addCustomer"
      else="deliveryWarning" />
</decision-state>
```

你可以看到，<decision-state>并不是独立完成工作的。<if>元素是决策状态的核心。这是表达式进行评估的地方，如果表达式结果

为true，流程将转移到then属性指定的状态中，如果结果为false，流程将会转移到else属性指定的状态中。

子流程状态

你可能不会将应用程序的所有逻辑写在一个方法中，而是将其分散到多个类、方法以及其他结构中。

同样，将流程分成独立的部分是个不错的主意。<subflow-state>允许在一个正在执行的流程中调用另一个流程。这类似于在一个方法中调用另一个方法。

<subflow-state>可以这样声明：

```
<subflow-state id="order" subflow="pizza/order">
  <input name="order" value="order"/>
  <transition on="orderCreated" to="payment" />
</subflow-state>
```

在这里，<input>元素用于传递订单对象作为子流程的输入。如果子流程结束的<end-state>状态ID为orderCreated，那么流程将会转移到名为payment的状态。

在这里，我有点超出进度了，我们还没有讨论到<end-state>元素和转移。我们很快就会在8.2.2小节介绍转移。对于结束状态，这正是接下来要介绍的。

结束状态

最后，所有的流程都要结束。这就是当流程转移到结束状态时所做的。<end-state>元素指定了流程的结束，它一般会是这样声明的：

```
<end-state id="customerReady" />
```

当到达<end-state>状态，流程会结束。接下来会发生什么取决于几个因素：

- 如果结束的流程是一个子流程，那调用它的流程将会从<subflow-state>处继续执行。<end-state>的ID将会用作事件触发从<subflow-state>开始的转移。
- 如果<end-state>设置了view属性，指定的视图将会被渲染。视图可以是相对于流程路径的视图模板，如果添加“externalRedirect:”前缀的话，将会重定向到流程外部的

页面，如果添加“flowRedirect:”将重定向到另一个流程中。
- 如果结束的流程不是子流程，也没有指定view属性，那这个流程只是会结束而已。浏览器最后将会加载流程的基本URL地址，当前已没有活动的流程，所以会开始一个新的流程实例。

需要意识到流程可能会有不止一个结束状态。子流程的结束状态ID确定了激活的事件，所以你可能会希望通过多种结束状态来结束子流程，从而能够在调用流程中触发不同的事件。即使不是在子流程中，也有可能在结束流程后，根据流程的执行情况有多个显示页面供选择。

现在，已经看完了流程中的各个状态，我们应当看一下流程是如何在状态间迁移的。让我们看看如何在流程中通过定义转移来完成道路铺设的。

8.2.2 转移

正如我在前面所提到的，转移连接了流程中的状态。流程中除结束状态之外的每个状态，至少都需要一个转移，这样就能够知道一旦这个状态完成时流程要去向哪里。状态可以有多个转移，分别对应于当前状态结束时可以执行的不同的路径。

转移使用<transition>元素来进行定义，它会作为各种状态元素（<action-state>、<view-state>、<subflow-state>）的子元素。最简单的形式就是<transition>元素在流程中指定下一个状态：

```
<transition to="customerReady" />
```

属性to用于指定流程的下一个状态。如果<transition>只使用了to属性，那这个转移就会是当前状态的默认转移选项，如果没有其他可用转移的话，就会使用它。

更常见的转移定义是基于事件的触发来进行的。在视图状态，事件通常会是用户采取的动作。在行为状态，事件是评估表达式得到的结果。而在子流程状态，事件取决于子流程结束状态的ID。在任意的事件中（这里没有任何歧义），你可以使用on属性来指定触发转移的事件：

```
<transition on="phoneEntered" to="lookupCustomer"/>
```

在本例中，如果触发了phoneEntered事件，流程将会进入lookupCustomer状态。

在抛出异常时，流程也可以进入另一个状态。例如，如果顾客的记录没有找到，你可能希望流程转移到一个展现注册表单的视图状态。以下的代码片段显示了这种类型的转移：

```
<transition
    on-exception=
        "com.springinaction.pizza.service.CustomerNotFoundException"
    to="registrationForm" />
```

属性on-exception类似于on属性，只不过它指定了要发生转移的异常而不是一个事件。在本示例中，CustomerNotFoundException异常将导致流程转移到registrationForm状态。

全局转移

在创建完流程之后，你可能会发现有一些状态使用了一些通用的转移。例如，如果在整个流程中到处都有如下<transition>的话，我一点也不感觉意外：

```
<transition on="cancel" to="endState" />
```

与其在多个状态中都重复通用的转移，我们可以将<transition>元素作为<global-transitions>的子元素，把它们定义为全局转移。例如：

```
<global-transitions>
  <transition on="cancel" to="endState" />
</global-transitions>
```

定义完这个全局转移后，流程中的所有状态都会默认拥有这个cancel转移。

我们已经讨论过了状态和转移。在我们开始编写流程之前，让我们看一下流程数据，这是Web流程三元素中的另一个成员。

8.2.3　流程数据

如果你曾经玩过那种老式的基于文字的冒险游戏的话，那么当从一个地方转移到另一个地方时，你会偶尔发现散布在周围的一些东西，你可以把它们捡起来并带上。有时候，你会马上需要一件东西。其他的

时候，你会在整个游戏过程中带着这些东西而不知道它们是做什么用的——直到你到达游戏结束的时候才会发现它是真正有用的。

在很多方面，流程与这些冒险游戏是很类似的。当流程从一个状态进行到另一个状态时，它会带走一些数据。有时候，这些数据只需要很短的时间（可能只要展现页面给用户）。有时候，这些数据会在整个流程中传递并在流程结束的时候使用。

声明变量

流程数据保存在变量中，而变量可以在流程的各个地方进行引用。它能够以多种方式创建。在流程中创建变量的最简单形式是使用<var>元素：

```
<var name="customer" class="com.springinaction.pizza.domain.Customer"/>
```

这里，创建了一个新的Customer实例并将其放在名为customer的变量中。这个变量可以在流程的任意状态进行访问。

作为行为状态的一部分或者作为视图状态的入口，你有可能会使用<evaluate>元素来创建变量。例如：

```
<evaluate result="viewScope.toppingsList"
    expression="T(com.springinaction.pizza.domain.Topping).asList()" />
```

在本例中，<evaluate>元素计算了一个表达式（SpEL表达式）并将结果放到了名为toppingsList的变量中，这个变量是视图作用域的（我们将会在稍后介绍关于作用域的更多概念）。

类似地，<set>元素也可以设置变量的值：

```
<set name="flowScope.pizza"
   value="new com.springinaction.pizza.domain.Pizza()" />
```

<set>元素与<evaluate>元素很类似，都是将变量设置为表达式计算的结果。这里，我们设置了一个流程作用域内的pizza变量，它的值是Pizza对象的新实例。

当我们在8.3小节开始构建真实工作的Web流程时，你会看到这些元素是如何具体应用在实际流程中的。但首先，让我们看一下变量的流程作用域、视图作用域以及其他的一些作用域是什么意思。

定义流程数据的作用域

流程中携带的数据会拥有不同的生命作用域和可见性，这取决于保存数据的变量本身的作用域。Spring Web Flow定义了五种不同作用域，如表8.2所示。

表8.2　**Spring Web Flow**的作用域

范　　围	生命作用域和可见性
Conversation	最高层级的流程开始时创建，在最高层级的流程结束时销毁。被最高层级的流程和其所有的子流程所共享
Flow	当流程开始时创建，在流程结束时销毁。只有在创建它的流程中是可见的
Request	当一个请求进入流程时创建，在流程返回时销毁
Flash	当流程开始时创建，在流程结束时销毁。在视图状态渲染后，它也会被清除
View	当进入视图状态时创建，当这个状态退出时销毁。只在视图状态内是可见的

当使用`<var>`元素声明变量时，变量始终是流程作用域的，也就是在定义变量的流程内有效。当使用`<set>`或`<evaluate>`的时候，作用域通过`name`或`result`属性的前缀指定。例如，将一个值赋给流程作用域的`theAnswer`变量：

```
<set name="flowScope.theAnswer" value="42"/>
```

到目前为止，我们已经看到了Web流程的所有原材料。是时候将其组装起来形成一个成熟且完整功能的Web流程了。当我们这样做的时候，请睁大你的眼睛观察，比如我是如何将数据存储在各作用域的变量中的。

8.3　组合起来：披萨流程

正如我在本章前面所提到的，我们将暂时不用Spittr应用程序。取而代之，我们被要求做一个在线的披萨订购应用，饥饿的Web访问者可以在这里订购他们所喜欢的意大利派。

实际上，订购披萨的过程可以很好地定义在一个流程中。我们首先从构建一个高层次的流程开始，它定义了订购披萨的整体过程。接下来，我们会将这个流程拆分成子流程，这些子流程在较低的层次定义了细节。

8.3.1　定义基本流程

一个新的披萨连锁店Spizza决定允许用户在线订购以减轻店面电话的压力。当顾客访问Spizza站点时，他们需要进行用户识别，选择一个或更多披萨添加到订单中，提供支付信息然后提交订单并等待热乎又新鲜的披萨送过来。图8.2阐述了这个流程。

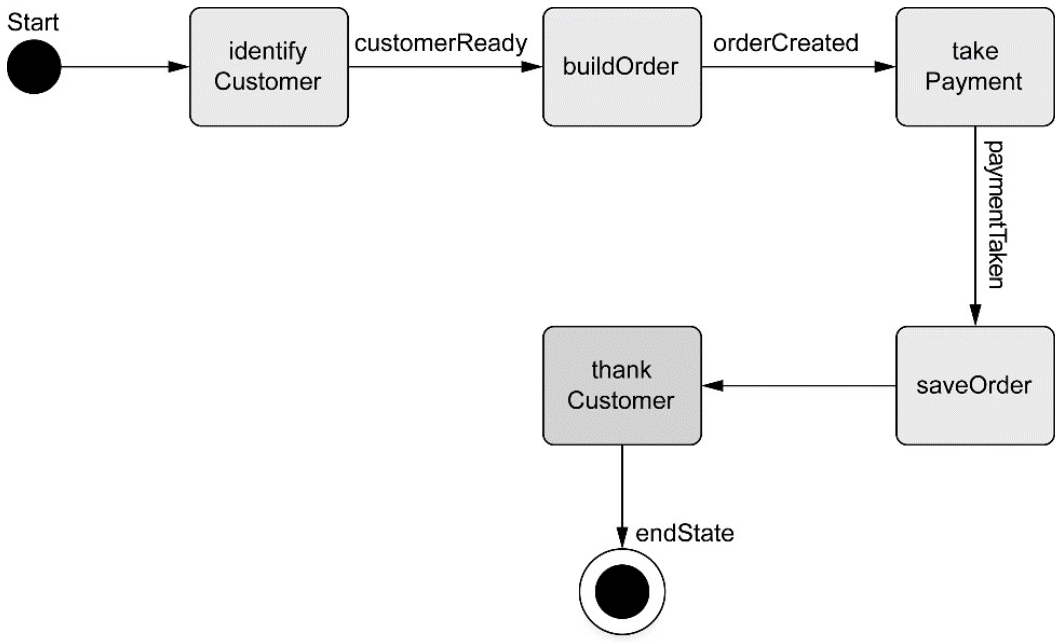

图8.2　将订购披萨的过程归结为一个简单的流程

图中的方框代表了状态而箭头代表了转移。你可以看到，订购披萨的整个流程很简单且是线性的。在Spring Web Flow中，表示这个流程是很容易的。使这个过程变得更有意思的就是前三个流程会比图中的简单方框更复杂。

以下的程序清单8.1展示了如何使用Spring Web Flow的XML流程定义来实现披萨订单的整体流程。

程序清单**8.1**　披萨订单流程定义为**Spring Web Flow**

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd">
  <var name="order"
       class="com.springinaction.pizza.domain.Order"/>
  <subflow-state id="identifyCustomer" subflow="pizza/customer">
    <output name="customer" value="order.customer"/>
    <transition on="customerReady" to="buildOrder" />
  </subflow-state>
  <subflow-state id="buildOrder" subflow="pizza/order">
    <input name="order" value="order"/>
    <transition on="orderCreated" to="takePayment" />
  </subflow-state>
  <subflow-state id="takePayment" subflow="pizza/payment">
    <input name="order" value="order"/>
    <transition on="paymentTaken" to="saveOrder"/>
  </subflow-state>
  <action-state id="saveOrder">
    <evaluate expression="pizzaFlowActions.saveOrder(order)" />
    <transition to="thankCustomer" />
  </action-state>
  <view-state id="thankCustomer">
    <transition to="endState" />
  </view-state>
  <end-state id="endState" />
  <global-transitions>
    <transition on="cancel" to="endState" />
  </global-transitions>
</flow>
```

调用顾客子流程
调用订单子流程
调用支付子流程
保存订单
感谢顾客
全局取消转移

在流程定义中，我们看到的第一件事就是order变量的声明。每次流程开始的时候，都会创建一个Order实例。Order类会带有关于订单的所有信息，包含顾客信息、订购的披萨列表以及支付详情，如下面所示。

程序清单**8.2**　**Order**带有披萨订单的所有细节信息

```java
package com.springinaction.pizza.domain;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
public class Order implements Serializable {
   private static final long serialVersionUID = 1L;
   private Customer customer;
   private List<Pizza> pizzas;
   private Payment payment;
   public Order() {
      pizzas = new ArrayList<Pizza>();
      customer = new Customer();
   }
   public Customer getCustomer() {
      return customer;
   }
   public void setCustomer(Customer customer) {
      this.customer = customer;
   }
   public List<Pizza> getPizzas() {
      return pizzas;
   }
   public void setPizzas(List<Pizza> pizzas) {
      this.pizzas = pizzas;
   }
   public void addPizza(Pizza pizza) {
      pizzas.add(pizza);
   }
   public float getTotal() {
      return 0.0f;
   }
   public Payment getPayment() {
      return payment;
   }
   public void setPayment(Payment payment) {
      this.payment = payment;
   }
}
```

流程定义的主要组成部分是流程的状态。默认情况下，流程定义文件中的第一个状态也会是流程访问中的第一个状态。在本例中，也就是identifyCustomer状态（一个子流程）。但是如果你愿意的话，你可以通过<flow>元素的start-state属性将任意状态指定为开始状态。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd"
  start-state="identifyCustomer">
...
</flow>
```

识别顾客、构造披萨订单以及支付这样的活动太复杂了，并不适合将其强行塞入一个状态。这是我们为何在后面将其单独定义为流程的原因。但是为了更好地整体了解披萨流程，这些活动都是以<subflow-state>元素来进行展现的。

流程变量order将在前三个状态中进行填充并在第四个状态中进行保

存。identifyCustomer子流程状态使用了<output>元素来填充order的customer属性，将其设置为顾客子流程收到的输出。buildOrder和takePayment状态使用了不同的方式，它们使用<input>将order流程变量作为输入，这些子流程就能在其内部填充order对象。

在订单得到顾客、一些披萨以及支付细节后，就可以对其进行保存了。saveOrder是处理这个任务的行为状态。它使用<evaluate>来调用ID为pizzaFlowActions的bean的saveOrder()方法，并将保存的订单对象传递进来。订单完成保存后，它会转移到thankCustomer。

thankCustomer状态是一个简单的视图状态，后台使用了“/WEB-INF/flows/pizza/ thankCustomer.jsp”这个JSP文件，如下所示：

程序清单**8.3**　感谢顾客订购的**JSP**视图

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />
  <head><title>Spizza</title></head>
  <body>
    <h2>Thank you for your order!</h2>
    <![CDATA[
    <a href='${flowExecutionUrl}&_eventId=finished'>Finish</a>   <-- 触发结束事件
    ]]>
    </body>
</html>
```

在“感谢”页面中，会感谢顾客的订购并为其提供一个完成流程的链接。这个链接是整个页面中最有意思的事情，因为它展示了用户与流程交互的唯一办法。

Spring Web Flow为视图的用户提供了一个flowExecutionUrl变量，它包含了流程的URL。结束链接将一个“_eventId”参数关联到URL上，以便回到Web流程时触发finished事件。这个事件将会让流程到达结束状态。

流程将会在结束状态完成。鉴于在流程结束后没有下一步做什么的具体信息，流程将会重新从identifyCustomer状态开始，以准备接受另一个披萨订单。

这涵盖了订购披萨的整体流程。但是这个流程并不仅仅是我们在代码清单8.1中所看到的这些。我们还需要定义identifyCustomer、buildOrder、takePayment这些状态的子流程。让我们从识别用户开始构建这些流程。

8.3.2 收集顾客信息

如果你曾经订购过披萨，你可能会知道流程。他们首先会询问你的电话号码。电话号码除了能够让送货司机在找不到你家的时候打电话给你，还可以作为你在这个披萨店的标识。如果你是回头客，他们可以使用这个电话号码来查找你的地址，这样他们就知道将你的订单派送到什么地方了。

对于一个新的顾客来讲，查询电话号码不会有什么结果。所以接下来，他们将询问你的地址。这样，披萨店的人就会知道你是谁以及将披萨送到哪里。但是在问你要哪种披萨之前，他们要确认你的地址在他们的配送范围之内。如果不在的话，你需要自己到店里并取走披萨。

在每个披萨订单开始前的提问和回答阶段可以用图8.3的流程图来表示。

这个流程比整体的披萨流程更有意思。这个流程不是线性的而是在好几个地方根据不同的条件有了分支。例如，在查找顾客后，流程可能结束（如果找到了顾客），也有可能转移到注册表单（如果没有找到顾客）。同样，在`checkDeliveryArea`状态，顾客有可能会被警告也有可能不被警告他们的地址在配送范围之外。

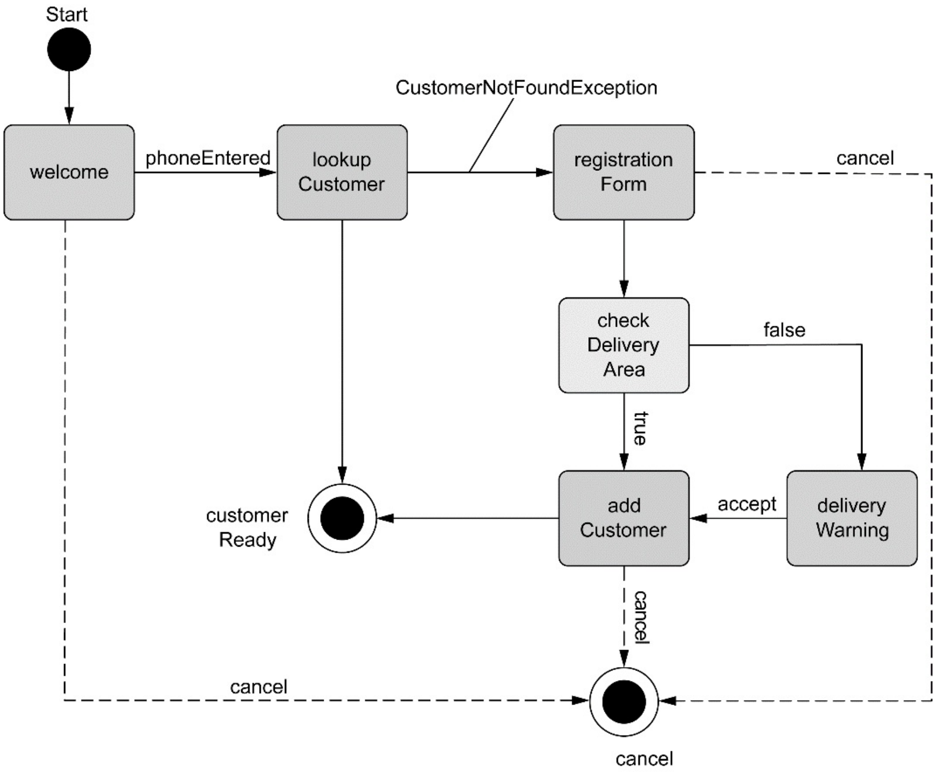

图8.3　识别顾客的流程比披萨流程有了更多的分支

以下的程序清单展示了识别顾客的流程定义。

程序清单**8.4**　使用**Web**流程来识别饥饿的披萨顾客

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd">
  <var name="customer"
       class="com.springinaction.pizza.domain.Customer"/>
  <view-state id="welcome">                               <— 欢迎顾客
     <transition on="phoneEntered" to="lookupCustomer"/>
  </view-state>
  <action-
     state id="lookupCustomer">                               <— 查找顾客
     <evaluate result="customer" expression=
   "pizzaFlowActions.lookupCustomer(requestParameters.phoneNumber)" />
    <transition to="registrationForm" on-exception=
        "com.springinaction.pizza.service.CustomerNotFoundException" />
    <transition to="customerReady" />
  </action-state>
  <view-state id="registrationForm" model="customer">   <— 注册新顾客
     <on-entry>
      <evaluate expression=
          "customer.phoneNumber = requestParameters.phoneNumber" />
    </on-entry>
    <transition on="submit" to="checkDeliveryArea" />          检查配
  </view-state>                                                     送区域
  <decision-state id="checkDeliveryArea">
     <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
        then="addCustomer"
        else="deliveryWarning"/>                            显示配
  </decision-state>                                          送警告
  <view-state id="deliveryWarning">
     <transition on="accept" to="addCustomer" />
  </view-state>                                            添加顾客
  <action-state id="addCustomer">
     <evaluate expression="pizzaFlowActions.addCustomer(customer)" />
    <transition to="customerReady" />
  </action-state>
  <end-state id="cancel" />
  <end-state id="customerReady">
    <output name="customer" />
  </end-state>
  <global-transitions>
    <transition on="cancel" to="cancel" />
  </global-transitions>
</flow>
```

这个流程包含了几个新的技巧，包括我们首次使用的`<decision-state>`元素。因为它是`pizza`流程的子流程，所以它也可以接受`Order`对象作为输入。

与前面一样，我们还是将这个流程的定义分解成一个个的状态，让我们从`welcome`状态开始。

询问电话号码

welcome状态是一个很简单的视图状态，它欢迎访问Spizza站点的顾客并要求他们输入电话号码。这个状态并没有什么特殊的。它有两个转移：如果从视图触发phoneEntered事件的话，转移会将流程定向到lookupCustomer，另外一个就是在全局转移中定义的用来响应cancel事件的cancel转移。

welcome状态的有趣之处在于视图本身。视图welcome定义在“/WEB-INF/flows/ pizza/customer/welcome.jspx”中，如下所示。

程序清单**8.5**　欢迎用户并询问他们的电话号码

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:form="http://www.springframework.org/tags/form">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />
  <head><title>Spizza</title></head>
  <body>
    <h2>Welcome to Spizza!!!</h2>
    <form:form>
      <input type="hidden" name="_flowExecutionKey"
             value="${flowExecutionKey}"/>           <-- 流程执行的 key
       <input type="text" name="phoneNumber"/><br/>
      <input type="submit" name="_eventId_phoneEntered"
             value="Lookup Customer" />              <-- 触发 phoneEntered 事件
    </form:form>
  </body>
</html>
```

这个简单的表单提示用户输入其电话号码。但是表单中有两个特殊的部分来驱动流程继续。

首先要注意的是隐藏的“_flowExecutionKey”输入域。当进入视图状态时，流程暂停并等待用户采取一些行为。赋予视图的流程执行key（flow execution key）就是一种返回流程的“回程票”（claim ticket）。当用户提交表单时，流程执行key会在“_flowExecutionKey”输入域中返回并在流程暂停的位置进行恢复。

还要注意的是提交按钮的名字。按钮名字的“_eventId_”部分是提供给Spring Web Flow的一个线索，它表明了接下来要触发事件。当点击这个按钮提交表单时，会触发phoneEntered事件进而转移到lookupCustomer。

查找顾客

当欢迎表单提交后，顾客的电话号码将包含在请求参数中并准备用于查询顾客。lookupCustomer状态的<evaluate>元素是查找发生

的地方。它将电话号码从请求参数中抽取出来并传递到pizzaFlowActions bean的lookupCustomer()方法中。

目前，lookupCustomer()的实现并不重要。只需知道它要么返回Customer对象，要么抛出CustomerNotFoundException异常。

在前一种情况下，Customer对象将会设置到customer变量中（通过result属性）并且默认的转移将把流程带到customerReady状态。但是如果不能找到顾客的话，将抛出CustomerNotFoundException并且流程被转移到registrationForm状态。

注册新顾客

registrationForm状态是要求用户填写配送地址的。就像我们之前看到的其他视图状态，它将被渲染成JSP。JSP文件如下所示。

程序清单**8.6**　注册新顾客

```
<html xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:form="http://www.springframework.org/tags/form">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />
  <head><title>Spizza</title></head>
  <body>
    <h2>Customer Registration</h2>
    <form:form commandName="customer">
      <input type="hidden" name="_flowExecutionKey"
             value="${flowExecutionKey}"/>
      <b>Phone number: </b><form:input path="phoneNumber"/><br/>
      <b>Name: </b><form:input path="name"/><br/>
      <b>Address: </b><form:input path="address"/><br/>
      <b>City: </b><form:input path="city"/><br/>
      <b>State: </b><form:input path="state"/><br/>
      <b>Zip Code: </b><form:input path="zipCode"/><br/>
      <input type="submit" name="_eventId_submit"
             value="Submit" />
      <input type="submit" name="_eventId_cancel"
             value="Cancel" />
    </form:form>
    </body>
</html>
```

这并非我们在流程中看到的第一个表单。welcome视图状态也为顾客展现了一个表单，那个表单很简单，并且只有一个输入域，从请求参数中获得输入域的值也很简单。但是注册表单就比较复杂了。

在这里不是通过请求参数一个个地处理输入域，而是以更好的方式将表单绑定到Customer对象上——让框架来做所有繁杂的工作。

检查配送区域

在顾客提供其地址后，我们需要确认他的住址在配送范围之内。如果Spizza不能派送给他们，那么我们要让顾客知道并建议他们自己到店面里取走披萨。

为了做出这个判断，我们使用了决策状态。决策状态checkDeliveryArea有一个<if>元素，它将顾客的邮政编码传递到pizzaFlowActions bean的checkDeliveryArea()方法中。这个方法将会返回一个Boolean值：如果顾客在配送区域内则为true，否则为false。

如果顾客在配送区域内的话，那流程转移到addCustomer状态。否则，顾客被带入到deliveryWarning视图状态。deliveryWarning背后的视图就是“/WEB-INF/flows/pizza/customer/deliveryWarning.jspx”，如下所示：

程序清单**8.7**　告知顾客不能将披萨配送到他们的地址

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />
  <head><title>Spizza</title></head>
  <body>
        <h2>Delivery Unavailable</h2>
        <p>The address is outside of our delivery area. You may
        still place the order, but you will need to pick it up
        yourself.</p>
        <![CDATA[
        <a href="${flowExecutionUrl}&_eventId=accept">
                                Continue, I'll pick up the order</a> |
        <a href="${flowExecutionUrl}&_eventId=cancel">Never mind</a>
        ]]>
  </body>
</html>
```

在deliveryWarning.jspx中与流程相关的两个关键点就是那两个链接，它们允许用户继续订单或者将其取消。通过使用与welcome状态相同的flowExecurtionUrl变量，这些链接分别触发流程中的accept或cancel事件。如果发送的是accept事件，那么流程会转移到addCustomer状态。否则，接下来会是全局的取消转移，子流程将会转移到cancel结束状态。

稍后我们将介绍结束状态。让我们先来看看addCustomer状态。

存储顾客数据

当流程抵达addCustomer状态时，用户已经输入了他们的地址。为

了将来使用，这个地址需要以某种方式存储起来（可能会存储在数据库中）。addCustomer状态有一个<evaluate>元素，它会调用pizzaFlowActions bean的addCustomer()方法，并将customer流程参数传递进去。

一旦这个过程完成，会执行默认的转移，流程将会转移到ID为customerReady的结束状态。

结束流程

一般来讲，流程的结束状态并不会那么有意思。但是这个流程中，它不仅仅只有一个结束状态，而是两个。当子流程完成时，它会触发一个与结束状态ID相同的流程事件。如果流程只有一个结束状态的话，那么它始终会触发相同的事件。但是如果有两个或更多的结束状态，流程能够影响到调用状态的执行方向。

当customer流程走完所有正常的路径后，它最终会到达ID为customerReady的结束状态。当调用它的披萨流程恢复时，它会接收到一个customerReady事件，这个事件将使得流程转移到buildOrder状态。

要注意的是customerReady结束状态包含了一个<output>元素。在流程中这个元素等同于Java中的return语句。它从子流程中传递一些数据到调用流程。在本示例中，<output>元素返回customer流程变量，这样在披萨流程中，就能够将identifyCustomer子流程的状态指定给订单。另一方面，如果在识别顾客流程的任意地方触发了cancel事件，将会通过ID为cancel的结束状态退出流程，这也会在披萨流程中触发cancel事件并导致转移（通过全局转移）到披萨流程的结束状态。

8.3.3 构建订单

在识别完顾客之后，主流程的下一件事情就是确定他们想要什么类型的披萨。订单子流程就是用于提示用户创建披萨并将其放入订单中的，如图8.4所示。

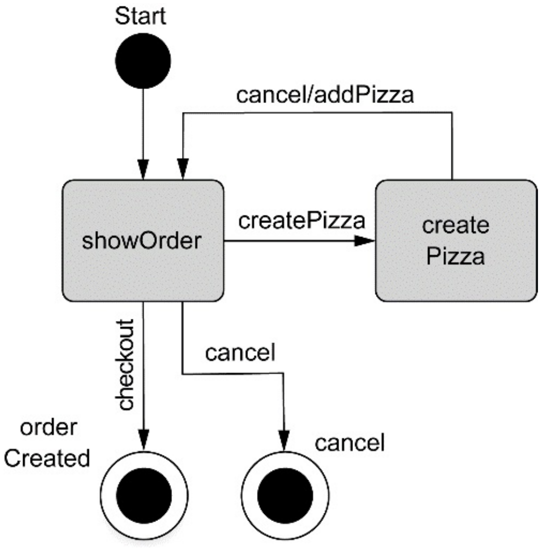

图8.4　通过订单子流程添加披萨

你可以看到，showOrder状态位于订单子流程的中心位置。这是用户进入这个流程时看到的第一个状态，它也是用户在添加披萨到订单后要转移到的状态。它展现了订单的当前状态并允许用户添加其他的披萨到订单中。

要添加披萨到订单时，流程会转移到createPizza状态。这是另外一个视图状态，允许用户选择披萨的尺寸和面饼上面的配料。在这里，用户可以添加或取消披萨，两种事件都会使流程转移回showOrder状态。

从showOrder状态，用户可能提交订单也可能取消订单。两种选择都会结束订单子流程，但是主流程会根据选择不同进入不同的执行路径。

如下显示了如何将图中所阐述的内容转变成Spring Web Flow定义。

程序清单**8.8**　订单子流程的视图状态，用于展示订单和添加披萨

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd">

  <input name="order" required="true" />                 <—— 接收 order 作为输入

  <view-state id="showOrder">                             <—— 展现 order 的状态
     <transition on="createPizza" to="createPizza" />
    <transition on="checkout" to="orderCreated" />
    <transition on="cancel" to="cancel" />
  </view-state>

  <view-
     state id="createPizza" model="flowScope.pizza">    <—— 创建披萨的状态
     <on-entry>
      <set name="flowScope.pizza"
          value="new com.springinaction.pizza.domain.Pizza()" />
      <evaluate result="viewScope.toppingsList"  expression=
           "T(com.springinaction.pizza.domain.Topping).asList()" />
    </on-entry>
    <transition on="addPizza" to="showOrder">
      <evaluate expression="order.addPizza(flowScope.pizza)" />
    </transition>
    <transition on="cancel" to="showOrder" />
  </view-state>

  <end-state id="cancel" /          <——————— 取消的结束状态

  <end-state id="orderCreated" />       <——————— 创建订单的结束状态

</flow>
```

这个子流程实际上会操作主流程创建的Order对象。因此，我们需要以某种方式将Order从主流程传到子流程。你可能还记得在程序清单8.1中我们使用了<input>元素来将Order传递进流程。在这里，我们使用它来接收Order对象。如果你觉得这个流程与Java中的方法有些类似地话，那这里使用的<input>元素实际上就定义了这个子流程的签名。这个流程需要一个名为order的参数。

接下来，我们会看到showOrder状态，它是一个基本的视图状态并具有三个不同的转移，分别用于创建披萨、提交订单以及取消订单。

createPizza状态更有意思一些。它的视图是一个表单，这个表单可以添加新的Pizza对象到订单中。<on-entry>元素添加了一个新的Pizza对象到流程作用域内，当表单提交时，表单的内容会填充到该对象中。需要注意的是，这个视图状态引用的model是流程作用域内的同一个Pizza对象。Pizza对象将绑定到创建披萨的表单中，如下所示。

程序清单**8.9**　通过将流程作用域的对象绑定到**HTML**表单，实现添加披萨到订单中

```
<div xmlns:form="http://www.springframework.org/tags/form"
     xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />
    <h2>Create Pizza</h2>
    <form:form commandName="pizza">
      <input type="hidden" name="_flowExecutionKey"
          value="${flowExecutionKey}"/>
      <b>Size: </b><br/>
  <form:radiobutton path="size"
                    label="Small (12-inch)" value="SMALL"/><br/>
  <form:radiobutton path="size"
                    label="Medium (14-inch)" value="MEDIUM"/><br/>
  <form:radiobutton path="size"
                    label="Large (16-inch)" value="LARGE"/><br/>
  <form:radiobutton path="size"
                    label="Ginormous (20-inch)" value="GINORMOUS"/>
      <br/>
      <br/>
      <b>Toppings: </b><br/>
      <form:checkboxes path="toppings" items="${toppingsList}"
                       delimiter="&lt;br/&gt;"/><br/><br/>
      <input type="submit" class="button"
          name="_eventId_addPizza" value="Continue"/>
      <input type="submit" class="button"
          name="_eventId_cancel" value="Cancel"/>
    </form:form>
</div>
```

当通过Continue按钮提交订单时，尺寸和配料选择将会绑定到Pizza对象中并且触发addPizza转移。与这个转移关联的<evaluate>元素表明在转移到showOrder状态之前，流程作用域内的Pizza对象将会传递给订单的addPizza()方法中。

有两种方法来结束这个流程。用户可以点击showOrder视图中的Cancel按钮或者Checkout按钮。这两种操作都会使流程转移到一个<end-state>。但是选择的结束状态id决定了退出这个流程时触发事件，进而最终确定了主流程的下一步行为。主流程要么基于cancel事件要么基于orderCreated事件进行状态转移。在前者情况下，外边的主流程会结束；在后者情况下，它将转移到takePayment子流程，这也是接下来我们要看的。

8.3.4 支付

吃免费披萨这事儿并不常见。如果Spizza披萨店让他们的顾客不提供支付信息就订购披萨的话，估计他们也维持不了多久。在披萨流程要结束的时候，最后的子流程提示用户输入他们的支付信息。这个简单的流程如图8.5所示。

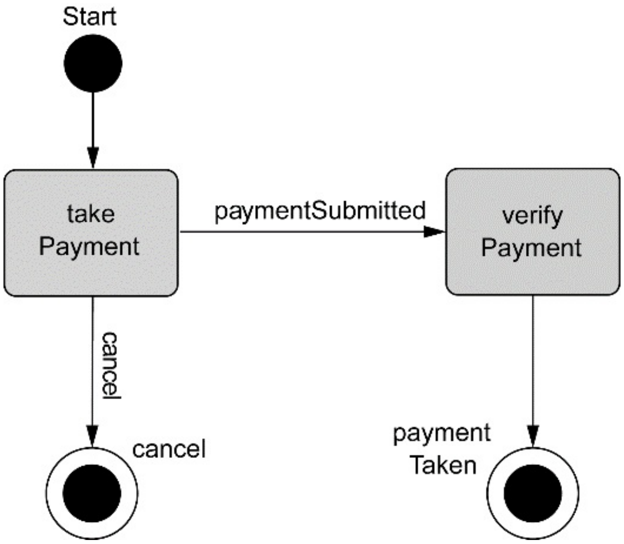

图8.5　订购披萨的最后一步是通过支付子流程让用户进行支付

像订单子流程一样，支付子流程也使用<input>元素接收一个Order对象作为输入。

你可以看到，进入支付子流程的时候，用户会到达takePayment状态。这是一个视图状态，在这里用户可以选择使用信用卡、支票或现金进行支付。提交支付信息后，将进入verifyPayment状态。这是一个行为状态，它将校验支付信息是否可以接受。

使用XML定义的支付流程如下所示：

程序清单**8.10**　支付子流程有一个视图状态和一个行为状态

```xml
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd">
  <input name="order" required="true"/>
  <view-state id="takePayment" model="flowScope.paymentDetails">
    <on-entry>
      <set name="flowScope.paymentDetails"
 value="new com.springinaction.pizza.domain.PaymentDetails()" />
      <evaluate result="viewScope.paymentTypeList" expression=
"T(com.springinaction.pizza.domain.PaymentType).asList()" />
    </on-entry>
    <transition on="paymentSubmitted" to="verifyPayment" />
    <transition on="cancel" to="cancel" />
  </view-state>
  <action-state id="verifyPayment">
    <evaluate result="order.payment" expression=
        "pizzaFlowActions.verifyPayment(flowScope.paymentDetails)" />
    <transition to="paymentTaken" />
  </action-state>

  <end-state id="cancel" />
  <end-state id="paymentTaken" />
</flow>
```

在流程进入takePayment视图时，<on-entry>元素将构建一个支付表单并使用SpEL表达式在流程作用域内创建一个PaymentDetails实例，这是支撑表单的对象。它也会创建视图作用域的paymentTypeList变量，这个变量是一个列表包含了PaymentType枚举（如程序清单8.11所示）的值。在这里，SpEL的T()操作用于获得PaymentType类，这样就可以调用静态的asList()方法。

程序清单**8.11**　**PaymentType**枚举定义了用户可用的支付选项

```java
package com.springinaction.pizza.domain;
import static org.apache.commons.lang.WordUtils.*;
import java.util.Arrays;
import java.util.List;
public enum PaymentType {
  CASH, CHECK, CREDIT_CARD;
  public static List<PaymentType> asList() {
    PaymentType[] all = PaymentType.values();
    return Arrays.asList(all);
  }
  @Override
  public String toString() {
    return capitalizeFully(name().replace('_', ' '));
  }
}
```

在面对支付表单的时候，用户可能提交支付也可能会取消。根据做出的选择，支付子流程将以名为paymentTaken或cancel的<end-state>结束。就像其他的子流程一样，不论哪种<end-state>都会结束子流程并将控制交给主流程。但是所采用<end-state>的id将决定主流程中接下来的转移。

现在，我们已经依次介绍了披萨流程及其子流程，并看到了Spring Web Flow的很多功能。在我们结束Spring Web Flow话题之前，让我们快速了解一下如何对流程及其状态的访问增加安全保护。

8.4 保护Web流程

在下一章中，我们将会看到如何使用Spring Security来保护Spring应用程序。但现在我们讨论的是Spring Web Flow，让我们快速地看一下Spring Web Flow是如何结合Spring Security支持流程级别的安全性的。

Spring Web Flow中的状态、转移甚至整个流程都可以借助<secured>元素实现安全性，该元素会作为这些元素的子元素。例如，为了保护对一个视图状态的访问，你可以这样使用<secured>：

```
<view-state id="restricted">
  <secured attributes="ROLE_ADMIN" match="all"/>
</view-state>
```

按照这里的配置，只有授予ROLE_ADMIN访问权限（借助attributes属性）的用户才能访问这个视图状态。attributes属性使用逗号分隔的权限列表来表明用户要访问指定状态、转移或流程所需要的权限。match属性可以设置为any或all。如果设置为any，那么用户必须至少具有一个attributes属性所列的权限。如果设置为all，那么用户必须具有所有的权限。你可能想知道用户如何具备<secured>元素所检验的权限，甚至最开始的时候用户是如何进行登录的？这些问题的答案将在第9章给出。

8.5 小结

并不是所有的Web应用程序都是自由访问的。有时候，必须对用户进行指引、询问适当的问题并基于他们的响应将其引导到特定页面。在这些情况下，应用程序不太像一个菜单选项而更像应用程序与用户之间的对话。

在本章中，我们介绍了Spring Web Flow，它是能够构建会话式应用程序的Web框架。在介绍的同时，我们构建了一个基于流程的披萨订单应用。我们先定义了应用程序的整体流程，从收集顾客信息开始到保存订单到系统中结束。

流程由多个状态和转移组成，它们定义了会话如何从一个状态到另一个状态。状态本身分为好多种：行为状态执行业务逻辑，视图状态涉及到流程中的用户，决策状态动态地引导流程执行，结束状态表明流程的结束，除此之外，还有子流程状态，它们自身是通过流程来定义的。

最后，我们看到如何限制具有特定权限的用户才能访问流程、状态或转移。但是，我们还没有介绍应用程序对用户的认证以及如何授予用户权限。这就是Spring Security能够发挥作用的地方了，而Spring Security就是我们第9章将要介绍的内容。

第9章　保护Web应用

本章内容：

- Spring Security介绍
- 使用Servlet规范中的Filter保护Web应用
- 基于数据库和LDAP进行认证

有一点不知道你是否在意过，那就是在电视剧中大多数人从不锁门？这是司空见惯的现象。在《*Seinfeld*》中，Kramer经常到Jerry的房间里并从他的冰箱里拿东西吃。在《*Friends*》中，很多剧中的角色经常不敲门就不假思索地进入别人的房间。有一次在伦敦，Ross甚至闯入Chandler的旅馆房间，差一点就撞见Chandler和Ross妹妹的私情。

在《*Leave it to Beaver*》热播的时代，人们不锁门这事儿并不值得大惊小怪。但是在这个隐私和安全被看得极其重要的年代，看到电视剧中的角色允许别人大摇大摆地进入自己的寓所或家中，实在让人难以置信。

现在，信息可能是我们最有价值的东西，一些不怀好意的人想尽办法试图偷偷进入不安全的应用程序来窃取我们的数据和身份信息。作为软件开发人员，我们必须采取措施来保护应用程序中的信息。无论你是通过用户名/密码来保护电子邮件账号，还是基于交易PIN来保护经纪账户，安全性都是绝大多数应用系统中的一个重要切面（aspect）。

我有意选择了“切面”这个词来描述应用系统的安全性。安全性是超越应用程序功能的一个关注点。应用系统的绝大部分内容都不应该参与到与自己相关的安全性处理中。尽管我们可以直接在应用程序中编写安全性功能相关的代码（这种情况并不少见），但更好的方式还是将安全性相关的关注点与应用程序本身的关注点进行分离。

如果你觉得安全性听上去好像是使用面向切面技术实现的，那你猜对了。在本章中，我们将使用切面技术来探索保护应用程序的方式。不过我们不必自己开发这些切面——我们将介绍Spring Security，这是一种基于Spring AOP和Servlet规范中的Filter实现的安全框架。

9.1 Spring Security简介

Spring Security是为基于Spring的应用程序提供声明式安全保护的安全性框架。Spring Security提供了完整的安全性解决方案，它能够在Web请求级别和方法调用级别处理身份认证和授权。因为基于Spring框架，所以Spring Security充分利用了依赖注入（dependency injection，DI）和面向切面的技术。

最初，Spring Security被称为Acegi Security。Acegi是一个强大的安全框架，但是它存在一个严重的问题：那就是需要大量的XML配置。我不会向你介绍这种复杂配置的细节。总之一句话，典型的Acegi配置有几百行XML是很常见的。

到了2.0版本，Acegi Security更名为Spring Security。但是2.0发布版本所带来的不仅仅是表面上名字的变化。为了在Spring中配置安全性，Spring Security引入了一个全新的、与安全性相关的XML命名空间。这个新的命名空间连同注解和一些合理的默认设置，将典型的安全性配置从几百行XML减少到十几行。Spring Security 3.0融入了SpEL，这进一步简化了安全性的配置。

它的最新版本为3.2，Spring Security从两个角度来解决安全性问题。它使用Servlet规范中的Filter保护Web请求并限制URL级别的访问。Spring Security还能够使用Spring AOP保护方法调用——借助于对象代理和使用通知，能够确保只有具备适当权限的用户才能访问安全保护的方法。

在本章中，我们将会关注如何将Spring Security用于Web层的安全性之中。在稍后的第14章中，我们会重新学习Spring Security，了解它如何保护方法的调用。

9.1.1 理解Spring Security的模块

不管你想使用Spring Security保护哪种类型的应用程序，第一件需要做的事就是将Spring Security模块添加到应用程序的类路径下。Spring Security 3.2分为11个模块，如表9.1所示。

表**9.1 Spring Security**被分成了**11**个模块

模　　块	描　　述

ACL	支持通过访问控制列表（access control list，ACL）为域对象提供安全性
切面（Aspects）	一个很小的模块，当使用Spring Security注解时，会使用基于AspectJ的切面，而不是使用标准的Spring AOP
CAS客户端（CAS Client）	提供与Jasig的中心认证服务（Central Authentication Service，CAS）进行集成的功能
配置（Configuration）	包含通过XML和Java配置Spring Security的功能支持
核心（Core）	提供Spring Security基本库
加密（Cryptography）	提供了加密和密码编码的功能
LDAP	支持基于LDAP进行认证
OpenID	支持使用OpenID进行集中式认证
Remoting	提供了对Spring Remoting的支持
标签库（Tag Library）	Spring Security的JSP标签库
Web	提供了Spring Security基于Filter的Web安全性支持

应用程序的类路径下至少要包含Core和Configuration这两个模块。Spring Security经常被用于保护Web应用，这显然也是Spittr应用的场景，所以我们还需要添加Web模块。同时我们还会用到Spring Security的JSP标签库，所以我们需要将这个模块也添加进来。

现在，我们已经为在Spring Security中进行安全性配置做好了准备。让我们看看如何使用Spring Security的XML命名空间。

9.1.2 过滤Web请求

Spring Security借助一系列Servlet Filter来提供各种安全性功能。你可能会想，这是否意味着我们需要在web.xml或WebApplicationInitializer中配置多个Filter呢？实际上，借助于Spring的小技巧，我们只需配置一个Filter就可以了。

DelegatingFilterProxy是一个特殊的Servlet Filter，它本身所做的工作并不多。只是将工作委托给一个javax.servlet.Filter实现类，这个实现类作为一个<bean>注册在Spring应用的上下文中，如图9.1所示。

图9.1 DelegatingFilterProxy把Filter的处理逻辑委托给Spring应用上下文中所定义的一个代理Filter bean

如果你喜欢在传统的web.xml中配置Servlet和Filter的话，可以使用<filter>元素，如下所示：

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
```

在这里，最重要的是<filter-name>设置成了springSecurityFilterChain。这是因为我们马上就会将Spring Security配置在Web安全性之中，这里会有一个名为springSecurityFilterChain的Filter bean，DelegatingFilterProxy会将过滤逻辑委托给它。

如果你希望借助WebApplicationInitializer以Java的方式来配置Delegating-FilterProxy的话，那么我们所需要做的就是创建一个扩展的新类：

```
package spitter.config;
import org.springframework.security.web.context.
                            AbstractSecurityWebApplicationInitializer;

public class SecurityWebInitializer
       extends AbstractSecurityWebApplicationInitializer {}
```

AbstractSecurityWebApplicationInitializer实现了WebApplication-Initializer，因此Spring会发现它，并用它在Web容器中注册DelegatingFilterProxy。尽管我们可以重载它的appendFilters()或insertFilters()方法来注册自己选择的Filter，但是要注册DelegatingFilterProxy的话，我们并不需要重载任何方法。

不管我们通过web.xml还是通过AbstractSecurityWebApplicationInitializer的子类来配置DelegatingFilterProxy，它都会拦截发往应用中的请求，并将请求委托给ID为springSecurityFilterChain bean。

springSecurityFilterChain本身是另一个特殊的Filter，它也被称为FilterChainProxy。它可以链接任意一个或多个其他的Filter。Spring Security依赖一系列Servlet Filter来提供不同的安全特性。但是，你几乎不需要知道这些细节，因为你不需要显式声明springSecurityFilterChain以及它所链接在一起的其他Filter。当我们启用Web安全性的时候，会自动创建这些Filter。

为了让Web安全性运行起来，我们创建一个最简单的安全性配置。

9.1.3 编写简单的安全性配置

在Spring Security的早期版本中（在其还被称为Acegi Security之时），为了在Web应用中启用简单的安全功能，我们需要编写上百行的XML配置。Spring Security 2.0提供了安全性相关的XML配置命名空间，让情况有了一些好转。

Spring 3.2引入了新的Java配置方案，完全不再需要通过XML来配置安全性功能了。如下的程序清单展现了Spring Security最简单的Java配置。

程序清单**9.1** 启用**Web**安全性功能的最简单配置

```
package spitter.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.
                                    configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.
                              configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity      <-- 启用Web安全性
public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

顾名思义，@EnableWebSecurity注解将会启用Web安全功能。但它本身并没有什么用处，Spring Security必须配置在一个实现了WebSecurityConfigurer的bean中，或者（简单起见）扩展WebSecurityConfigurerAdapter。在Spring应用上下文中，任何实现了WebSecurityConfigurer的bean都可以用来配置Spring Security，但是最为简单的方式还是像程序清单9.1那样扩展WebSecurityConfigurer Adapter类。

@EnableWebSecurity可以启用任意Web应用的安全性功能，不过，如果你的应用碰巧是使用Spring MVC开发的，那么就应该考虑使用@EnableWebMvcSecurity替代它，如程序清单9.2所示。

程序清单**9.2** 为**Spring MVC**启用**Web**安全性功能的最简单配置

```
package spitter.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.
                              configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.annotation.web.servlet.
                                      configuration.EnableWebMvcSecurity;

@Configuration
@EnableWebMvcSecurity
 public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

启用Spring MVC安全性

除了其他的内容以外，@EnableWebMvcSecurity注解还配置了一个Spring MVC参数解析解析器（argument resolver），这样的话处理器方法就能够通过带有@AuthenticationPrincipal注解的参数获得认证用户的principal（或username）。它同时还配置了一个bean，在使用Spring表单绑定标签库来定义表单时，这个bean会自动添加一个隐藏的跨站请求伪造（cross-site request forgery，CSRF）token输入域。

看起来似乎并没有做太多的事情，但程序清单9.1和9.2中的配置类会给应用产生很大的影响。其中任何一种配置都会将应用严格锁定，导致没有人能够进入该系统了！

尽管不是严格要求的，但我们可能希望指定Web安全的细节，这要通过重载WebSecurityConfigurerAdapter中的一个或多个方法来实现。我们可以通过重载WebSecurityConfigurerAdapter的三个configure()方法来配置Web安全性，这个过程中会使用传递进来的参数设置行为。表9.2描述了这三个方法。

表**9.2** 重载**WebSecurityConfigurerAdapter**的**configure()**方法

方　　法	描　　述
`configure(WebSecurity)`	通过重载，配置Spring Security的Filter链
`configure(HttpSecurity)`	通过重载，配置如何通过拦截器保护请求
`configure(AuthenticationManagerBuilder)`	通过重载，配置user-detail服务

让我们重新看一下程序清单9.2，可以看到它没有重写上述三个`configure()`方法中的任何一个，这就说明了为什么应用现在是被锁定的。尽管对于我们的需求来讲默认的Filter链是不错的，但是默认的`configure(HttpSecurity)`实际上等同于如下所示：

```
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .anyRequest().authenticated()
      .and()
    .formLogin().and()
    .httpBasic();
}
```

这个简单的默认配置指定了该如何保护HTTP请求，以及客户端认证用户的方案。通过调用`authorizeRequests()`和`anyRequest().authenticated()`就会要求所有进入应用的HTTP请求都要进行认证。它也配置Spring Security支持基于表单的登录以及HTTP Basic方式的认证。

同时，因为我们没有重载`configure(AuthenticationManagerBuilder)`方法，所以没有用户存储支撑认证过程。没有用户存储，实际上就等于没有用户。所以，在这里所有的请求都需要认证，但是没有人能够登录成功。

为了让Spring Security满足我们应用的需求，还需要再添加一点配置。具体来讲，我们需要：

- 配置用户存储；
- 指定哪些请求需要认证，哪些请求不需要认证，以及所需要的权限；

- 提供一个自定义的登录页面，替代原来简单的默认登录页。

除了Spring Security的这些功能，我们可能还希望基于安全限制，有选择性地在Web视图上显示特定的内容。

但首先，我们看一下如何在认证的过程中配置访问用户数据的服务。

9.2 选择查询用户详细信息的服务

假如你计划去一个独家经营的饭店享受一顿晚餐，当然，你会提前几周预订，保证到时候能有一个位置。当到达饭店的时候，你会告诉服务员你的名字。但令人遗憾的是，里面并没有你的预订记录。美好的夜晚眼看就要泡汤了。但是没有人会如此轻易地放弃，你会要求服务员再次确认预订名单。此时，事情变得有些怪异了。

服务员说没有预订名单。你的名字不在名单上——名单上没有任何人——因为根本就不存在这么个名单。这就解释了为什么位置是空的，但我们却进不去。几周后，我们也就明白这家饭店为何最终会关门大吉，被一家墨西哥美食店所代替。

这也是此时我们应用程序的现状。我们没有办法进入应用，即便用户认为他们应该能够登录进去，但实际上却没有允许他们访问应用的数据记录。因为缺少用户存储，现在的应用程序太封闭了，变得不可用。

我们所需要的是用户存储，也就是用户名、密码以及其他信息存储的地方，在进行认证决策的时候，会对其进行检索。

好消息是，Spring Security非常灵活，能够基于各种数据存储来认证用户。它内置了多种常见的用户存储场景，如内存、关系型数据库以及LDAP。但我们也可以编写并插入自定义的用户存储实现。

借助Spring Security的Java配置，我们能够很容易地配置一个或多个数据存储方案。那我们就从最简单的开始：在内存中维护用户存储。

9.2.1 使用基于内存的用户存储

因为我们的安全配置类扩展了`WebSecurityConfigurerAdapter`，因此配置用户存储的最简单方式就是重载`configure()`方法，并以

AuthenticationManagerBuilder作为传入参数。AuthenticationManagerBuilder有多个方法可以用来配置Spring Security对认证的支持。通过inMemoryAuthentication()方法，我们可以启用、配置并任意填充基于内存的用户存储。

例如，在如程序清单9.3中，SecurityConfig重载了configure()方法，并使用两个用户来配置内存用户存储。

程序清单**9.3**　配置**Spring Security**使用内存用户存储

```
package spitter.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.
                authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.
                         configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.annotation.web.servlet.
                              configuration.EnableWebMvcSecurity;

@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(AuthenticationManagerBuilder auth)
                                              throws Exception {
    auth
      .inMemoryAuthentication()      // 启用内存用户存储
        .withUser("user").password("password").roles("USER").and()
        .withUser("admin").password("password").roles("USER", "ADMIN");
  }

}
```

我们可以看到，configure()方法中的AuthenticationManagerBuilder使用构造者风格的接口来构建认证配置。通过简单地调用inMemoryAuthentication()就能启用内存用户存储。但是我们还需要有一些用户，否则的话，这和没有用户并没有什么区别。

因此，我们需要调用withUser()方法为内存用户存储添加新的用户，这个方法的参数是username。withUser()方法返回的是UserDetailsManagerConfigurer.UserDetailsBuilder，这个对象提供了多个进一步配置用户的方法，包括设置用户密码的password()方法以及为给定用户授予一个或多个角色权限的roles()方法。

在程序清单9.3中，我们添加了两个用户，“user”和“admin”，密码均为“password”。“user”用户具有USER角色，而“admin”用户具有ADMIN和USER两个角色。我们可以看到，and()方法能够将多个用户

的配置连接起来。

除了password()、roles()和and()方法以外，还有其他的几个方法可以用来配置内存用户存储中的用户信息。表9.3描述了UserDetailsManagerConfigurer.UserDetailsBuilder对象所有可用的方法。

需要注意的是，roles()方法是authorities()方法的简写形式。roles()方法所给定的值都会添加一个“ROLE_”前缀，并将其作为权限授予给用户。实际上，如下的用户配置与程序清单9.3是等价的：

```
auth
.inMemoryAuthentication()
  .withUser("user").password("password")
                              .authorities("ROLE_USER").and()
  .withUser("admin").password("password")
                              .authorities("ROLE_USER", "ROLE_ADMIN");
```

表**9.3**　配置用户详细信息的方法

方　　法	描　　述
accountExpired(boolean)	定义账号是否已经过期
accountLocked(boolean)	定义账号是否已经锁定
and()	用来连接配置
authorities(GrantedAuthority...)	授予某个用户一项或多项权限
authorities(List<? extends GrantedAuthority>)	授予某个用户一项或多项权限
authorities(String...)	授予某个用户一项或多项权限
credentialsExpired(boolean)	定义凭证是否已经过期
disabled(boolean)	定义账号是否已被禁用

password(String)	定义用户的密码
roles(String...)	授予某个用户一项或多项角色

对于调试和开发人员测试来讲，基于内存的用户存储是很有用的，但是对于生产级别的应用来讲，这就不是最理想的可选方案了。为了用于生产环境，通常最好将用户数据保存在某种类型的数据库之中。

9.2.2 基于数据库表进行认证

用户数据通常会存储在关系型数据库中，并通过JDBC进行访问。为了配置Spring Security使用以JDBC为支撑的用户存储，我们可以使用jdbcAuthentication()方法，所需的最少配置如下所示：

```
@Autowired
DataSource dataSource;

@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                   throws Exception {
  auth
    .jdbcAuthentication()
      .dataSource(dataSource);
}
```

我们必须要配置的只是一个DataSource，这样的话，就能访问关系型数据库了。在这里，DataSource是通过自动装配的技巧得到的。

重写默认的用户查询功能

尽管默认的最少配置能够让一切运转起来，但是它对我们的数据库模式有一些要求。它预期存在某些存储用户数据的表。更具体来说，下面的代码片段来源于Spring Security内部，这块代码展现了当查找用户信息时所执行的SQL查询语句：

```
public static final String DEF_USERS_BY_USERNAME_QUERY =
        "select username,password,enabled " +
        "from users " +
        "where username = ?";
public static final String DEF_AUTHORITIES_BY_USERNAME_QUERY =
        "select username,authority " +
        "from authorities " +
        "where username = ?";
public static final String DEF_GROUP_AUTHORITIES_BY_USERNAME_QUERY =
        "select g.id, g.group_name, ga.authority " +
        "from groups g, group_members gm, group_authorities ga " +
        "where gm.username = ? " +
        "and g.id = ga.group_id " +
        "and g.id = gm.group_id";
```

在第一个查询中，我们获取了用户的用户名、密码以及是否启用的信息，这些信息会用来进行用户认证。接下来的查询查找了用户所授予的权限，用来进行鉴权，最后一个查询中，查找了用户作为群组的成员所授予的权限。

如果你能够在数据库中定义和填充满足这些查询的表，那么基本上就不需要你再做什么额外的事情了。但是，也有可能你的数据库与上面所述并不一致，那么你就会希望在查询上有更多的控制权。如果是这样的话，我们可以按照如下的方式配置自己的查询：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                    throws Exception {
  auth
    .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
        "select username, password, true " +
        "from Spitter where username=?")
      .authoritiesByUsernameQuery(
        "select username, 'ROLE_USER' from Spitter where username=?");
}
```

在本例中，我们只重写了认证和基本权限的查询语句，但是通过调用`group-AuthoritiesByUsername()`方法，我们也能够将群组权限重写为自定义的查询语句。

将默认的SQL查询替换为自定义的设计时，很重要的一点就是要遵循查询的基本协议。所有查询都将用户名作为唯一的参数。认证查询会选取用户名、密码以及启用状态信息。权限查询会选取零行或多行包含该用户名及其权限信息的数据。群组权限查询会选取零行或多行数据，每行数据中都会包含群组ID、群组名称以及权限。

使用转码后的密码

看一下上面的认证查询，它会预期用户密码存储在了数据库之中。这里唯一的问题在于如果密码明文存储的话，会很容易受到黑客的窃取。但是，如果数据库中的密码进行了转码的话，那么认证就会失败，因为它与用户提交的明文密码并不匹配。

为了解决这个问题，我们需要借助`passwordEncoder()`方法指定一个密码转码器（encoder）：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                throws Exception {
  auth
    .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
        "select username, password, true " +
        "from Spitter where username=?")
      .authoritiesByUsernameQuery(
        "select username, 'ROLE_USER' from Spitter where username=?")
      .passwordEncoder(new StandardPasswordEncoder("53cr3t"));
}
```

passwordEncoder()方法可以接受Spring Security中PasswordEncoder接口的任意实现。Spring Security的加密模块包括了三个这样的实现：BCryptPasswordEncoder、NoOpPasswordEncoder和StandardPasswordEncoder。

上述的代码中使用了StandardPasswordEncoder，但是如果内置的实现无法满足需求时，你可以提供自定义的实现。PasswordEncoder接口非常简单：

```
public interface PasswordEncoder {
  String encode(CharSequence rawPassword);
  boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

不管你使用哪一个密码转码器，都需要理解的一点是，数据库中的密码是永远不会解码的。所采取的策略与之相反，用户在登录时输入的密码会按照相同的算法进行转码，然后再与数据库中已经转码过的密码进行对比。这个对比是在PasswordEncoder的matches()方法中进行的。

9.2.3 基于LDAP进行认证

为了让Spring Security使用基于LDAP的认证，我们可以使用ldapAuthentication()方法。这个方法在功能上类似于jdbcAuthentication()，只不过是LDAP版本。如下的configure()方法展现了LDAP认证的简单配置：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                throws Exception {
  auth
    .ldapAuthentication()
      .userSearchFilter("(uid={0})")
      .groupSearchFilter("member={0}");
}
```

方法userSearchFilter()和groupSearchFilter()用来为基础LDAP查询提供过滤条件，它们分别用于搜索用户和组。默认情况下，对于用户和组的基础查询都是空的，也就是表明搜索会在LDAP层级结构的根开始。但是我们可以通过指定查询基础来改变这个默认行为：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                   throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}");
}
```

userSearchBase()属性为查找用户提供了基础查询。同样，groupSearchBase()为查找组指定了基础查询。我们声明用户应该在名为people的组织单元下搜索而不是从根开始。而组应该在名为groups的组织单元下搜索。

配置密码比对

基于LDAP进行认证的默认策略是进行绑定操作，直接通过LDAP服务器认证用户。另一种可选的方式是进行比对操作。这涉及将输入的密码发送到LDAP目录上，并要求服务器将这个密码和用户的密码进行比对。因为比对是在LDAP服务器内完成的，实际的密码能保持私密。

如果你希望通过密码比对进行认证，可以通过声明passwordCompare()方法来实现：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                   throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare();
}
```

默认情况下，在登录表单中提供的密码将会与用户的LDAP条目中的userPassword属性进行比对。如果密码被保存在不同的属性中，可以通过passwordAttribute()方法来声明密码属性的名称：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare()
      .passwordEncoder(new Md5PasswordEncoder())
      .passwordAttribute("passcode");
}
```

在本例中，我们指定了要与给定密码进行比对的是“passcode”属性。另外，我们还可以指定密码转码器。在进行服务器端密码比对时，有一点非常好，那就是实际的密码在服务器端是私密的。但是进行尝试的密码还是需要通过线路传输到LDAP服务器上，这可能会被黑客所拦截。为了避免这一点，我们可以通过调用passwordEncoder()方法指定加密策略。

在本示例中，密码会进行MD5加密。这需要LDAP服务器上密码也使用MD5进行加密。

引用远程的**LDAP**服务器

到目前为止，我们忽略的一件事就是LDAP和实际的数据在哪里。我们很开心地配置Spring使用LDAP服务器进行认证，但是服务器在哪里呢？

默认情况下，Spring Security的LDAP认证假设LDAP服务器监听本机的33389端口。但是，如果你的LDAP服务器在另一台机器上，那么可以使用contextSource()方法来配置这个地址：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
  auth
  .ldapAuthentication()
    .userSearchBase("ou=people")
    .userSearchFilter("(uid={0})")
    .groupSearchBase("ou=groups")
    .groupSearchFilter("member={0}")
    .contextSource()
      .url("ldap://habuma.com:389/dc=habuma,dc=com");
}
```

contextSource()方法会返回一个ContextSourceBuilder对象，这个对象除了其他功能以外，还提供了url()方法用来指定LDAP服务器的地址。

配置嵌入式的**LDAP**服务器

如果你没有现成的LDAP服务器供认证使用，Spring Security还为我们提供了嵌入式的LDAP服务器。我们不再需要设置远程LDAP服务器的URL，只需通过`root()`方法指定嵌入式服务器的根前缀就可以了：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
  auth
  .ldapAuthentication()
    .userSearchBase("ou=people")
    .userSearchFilter("(uid={0})")
    .groupSearchBase("ou=groups")
    .groupSearchFilter("member={0}")
    .contextSource()
      .root("dc=habuma,dc=com");
}
```

当LDAP服务器启动时，它会尝试在类路径下寻找LDIF文件来加载数据。LDIF（LDAP Data Interchange Format，LDAP数据交换格式）是以文本文件展现LDAP数据的标准方式。每条记录可以有一行或多行，每项包含一个名值对。记录之间通过空行进行分割。

如果你不想让Spring从整个根路径下搜索LDIF文件的话，那么可以通过调用`ldif()`方法来明确指定加载哪个LDIF文件：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
  auth
  .ldapAuthentication()
    .userSearchBase("ou=people")
    .userSearchFilter("(uid={0})")
    .groupSearchBase("ou=groups")
    .groupSearchFilter("member={0}")
    .contextSource()
      .root("dc=habuma,dc=com")
      .ldif("classpath:users.ldif");
}
```

在这里，我们明确要求LDAP服务器从类路径根目录下的users.ldif文件中加载内容。如果你比较好奇的话，如下就是一个包含用户数据LDIF文件，我们可以使用它来加载嵌入式LDAP服务器：

```
dn: ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups
dn: ou=people,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people
dn: uid=habuma,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Craig Walls
sn: Walls
uid: habuma
userPassword: password
dn: uid=jsmith,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: John Smith
sn: Smith
uid: jsmith
userPassword: password
dn: cn=spittr,ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: groupOfNames
cn: spittr
member: uid=habuma,ou=people,dc=habuma,dc=com
```

Spring Security内置的用户存储非常便利，并且涵盖了最为常用的用户场景。但是，如果你的认证需求不是那么通用的话，那么就需要创建并配置自定义的用户详细信息服务了。

9.2.4 配置自定义的用户服务

假设我们需要认证的用户存储在非关系型数据库中，如Mongo或Neo4j，在这种情况下，我们需要提供一个自定义的UserDetailsService接口实现。

UserDetailsService接口非常简单：

```
public interface UserDetailsService {
  UserDetails loadUserByUsername(String username)
                                   throws UsernameNotFoundException;
}
```

我们所需要做的就是实现loadUserByUsername()方法，根据给定的用户名来查找用户。loadUserByUsername()方法会返回代表给定用户的UserDetails对象。如下的程序清单展现了一个UserDetailsService的实现，它会从给定的SpitterRepository实现中查找用户。

程序清单**9.4**　从**SpitterRepository**中查找**UserDetails**对象

```java
package spittr.security;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.
                                            SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.
                                                UserDetailsService;
import org.springframework.security.core.userdetails.
                                         UsernameNotFoundException;
import spittr.Spitter;
import spittr.data.SpitterRepository;

public class SpitterUserService implements UserDetailsService {

  private final SpitterRepository spitterRepository;

  public SpitterUserService(SpitterRepository spitterRepository) {   // 注入 Spitter-Repository
     this.spitterRepository = spitterRepository;
  }

  @Override
  public UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException {
    Spitter spitter = spitterRepository.findByUsername(username);   // 查找 Spitter
     if (spitter != null) {
      List<GrantedAuthority> authorities =
          new ArrayList<GrantedAuthority>();
      authorities.add(new SimpleGrantedAuthority("ROLE_SPITTER"));   // 创建权限列表

      return new User(                                               // 返回 User
           spitter.getUsername(),
          spitter.getPassword(),
          authorities);
    }

     throw new UsernameNotFoundException(
         "User '" + username + "' not found.");
  }

}
```

`SpitterUserService`有意思的地方在于它并不知道用户数据存储在什么地方。设置进来的`SpitterRepository`能够从关系型数据库、文档数据库或图数据中查找`Spitter`对象，甚至可以伪造一个。`SpitterUserService`不知道也不会关心底层所使用的数据存储。它只是获得Spitter对象，并使用它来创建`User`对象。（`User`是`UserDetails`的具体实现。）

为了使用`SpitterUserService`来认证用户，我们可以通过`userDetailsService()`方法将其设置到安全配置中：

```
@Autowired
SpitterRepository spitterRepository;

@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                    throws Exception {
  auth
    .userDetailsService(new SpitterUserService(spitterRepository));
}
```

userDetailsService()方法（类似于jdbcAuthentication()、ldapAuthentication以及inMemoryAuthentication()）会配置一个用户存储。不过，这里所使用的不是Spring所提供的用户存储，而是使用UserDetailsService的实现。

另外一种值得考虑的方案就是修改Spitter，让其实现UserDetails。这样的话，loadUserByUsername()就能直接返回Spitter对象了，而不必再将它的值复制到User对象中。

9.3 拦截请求

在前面的9.1.3小节中，我们看到一个特别简单的Spring Security配置，在这个默认的配置中，会要求所有请求都要经过认证。有些人可能会说，过多的安全性总比安全性太少要好。但也有一种说法就是要适量地应用安全性。

在任何应用中，并不是所有的请求都需要同等程度地保护。有些请求需要认证，而另一些可能并不需要。有些请求可能只有具备特定权限的用户才能访问，没有这些权限的用户会无法访问。

例如，考虑Spittr应用的请求。首页当然是公开的，不需要进行保护。类似地，因为所有的Spittle都是公开的，所以展现Spittle的页面不需要安全性。但是，创建Spittle的请求只有认证用户才能执行。同样，尽管用户基本信息页面是公开的，不需要认证，但是，如果要处理“/spitters/me”请求，并展现当前用户的基本信息时，那么就需要进行认证，从而确定要展现谁的信息。

对每个请求进行细粒度安全性控制的关键在于重载configure(HttpSecurity)方法。如下的代码片段展现了重载的configure(HttpSecurity)方法，它为不同的URL路径有选择地应用安全性：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/spitters/me").authenticated()
      .antMatchers(HttpMethod.POST, "/spittles").authenticated()
      .anyRequest().permitAll();
}
```

configure()方法中得到的HttpSecurity对象可以在多个方面配置HTTP的安全性。在这里，我们首先调用authorizeRequests()，然后调用该方法所返回的对象的方法来配置请求级别的安全性细节。其中，第一次调用antMatchers()指定了对"/spitters/me"路径的请求需要进行认证。第二次调用antMatchers()更为具体，说明对"/spittles"路径的HTTP POST请求必须要经过认证。最后对anyRequests()的调用中，说明其他所有的请求都是允许的，不需要认证和任何的权限。

antMatchers()方法中设定的路径支持Ant风格的通配符。在这里我们并没有这样使用，但是也可以使用通配符来指定路径，如下所示：

```
.antMatchers("/spitters/**").authenticated();
```

我们也可以在一个对antMatchers()方法的调用中指定多个路径：

```
.antMatchers("/spitters/**", "/spittles/mine").authenticated();
```

antMatchers()方法所使用的路径可能会包括Ant风格的通配符，而regexMatchers()方法则能够接受正则表达式来定义请求路径。例如，如下代码片段所使用的正则表达式与"/spitters/**"（Ant风格）功能是相同的：

```
.regexMatchers("/spitters/.*").authenticated();
```

除了路径选择，我们还通过authenticated()和permitAll()来定义该如何保护路径。authenticated()要求在执行该请求时，必须已经登录了应用。如果用户没有认证的话，Spring Security的Filter将会捕获该请求，并将用户重定向到应用的登录页面。同时，permitAll()方法允许请求没有任何的安全限制。

除了authenticated()和permitAll()以外，还有其他的一些方法能够用来定义该如何保护请求。表9.4描述了所有可用的方案。

表**9.4**　用来定义如何保护路径的配置方法

方　　法	能够做什么
access(String)	如果给定的SpEL表达式计算结果为true，就允许访问
anonymous()	允许匿名用户访问
authenticated()	允许认证过的用户访问
denyAll()	无条件拒绝所有访问
fullyAuthenticated()	如果用户是完整认证的话（不是通过Remember-me功能认证的），就允许访问
hasAnyAuthority(String...)	如果用户具备给定权限中的某一个的话，就允许访问
hasAnyRole(String...)	如果用户具备给定角色中的某一个的话，就允许访问
hasAuthority(String)	如果用户具备给定权限的话，就允许访问
hasIpAddress(String)	如果请求来自给定IP地址的话，就允许访问
hasRole(String)	如果用户具备给定角色的话，就允许访问
not()	对其他访问方法的结果求反
permitAll()	无条件允许访问
rememberMe()	如果用户是通过Remember-me功能认证的，就允许访问

通过使用表9.4中的方法，我们所配置的安全性能够不仅仅限于认证用户。例如，我们可以修改之前的`configure()`方法，要求用户不仅需要认证，还要具备ROLE_SPITTER权限：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/spitters/me").hasAuthority("ROLE_SPITTER")
      .antMatchers(HttpMethod.POST, "/spittles")
                              .hasAuthority("ROLE_SPITTER")
      .anyRequest().permitAll();
}
```

作为替代方案，我们还可以使用`hasRole()`方法，它会自动使用“`ROLE_`”前缀：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/spitter/me").hasRole("SPITTER")
      .antMatchers(HttpMethod.POST, "/spittles").hasRole("SPITTER")
      .anyRequest().permitAll();
}
```

我们可以将任意数量的`antMatchers()`、`regexMatchers()`和`anyRequest()`连接起来，以满足Web应用安全规则的需要。但是，我们需要知道，这些规则会按照给定的顺序发挥作用。所以，很重要的一点就是将最为具体的请求路径放在前面，而最不具体的路径（如`anyRequest()`）放在最后面。如果不这样做的话，那不具体的路径配置将会覆盖掉更为具体的路径配置。

9.3.1　使用Spring表达式进行安全保护

表9.4中的大多数方法都是一维的，也就是说我们可以使用`hasRole()`限制某个特定的角色，但是我们不能在相同的路径上同时通过`hasIpAddress()`限制特定的IP地址。

另外，除了表9.4定义的方法以外，我们没有办法使用其他的条件。如果我们希望限制某个角色只能在星期二进行访问的话，该怎么办呢？

在第3章中，我们看到了如何使用Spring表达式语言（Spring Expression Language，SpEL），将其作为装配bean属性的高级技术。借助`access()`方法，我们也可以将SpEL作为声明访问限制的一种方式。例如，如下就是使用SpEL表达式来声明具有“ROLE_SPITTER”角

色才能访问“/spitter/me”URL：

```
.antMatchers("/spitter/me").access("hasRole('ROLE_SPITTER')")
```

这个对“/spitter/me”的安全限制与开始时的效果是等价的，只不过这里使用了SpEL来描述安全规则。如果当前用户被授予了给定角色的话，那`hasRole()`表达式的计算结果就为`true`。

让SpEL更强大的原因在于，`hasRole()`仅是Spring支持的安全相关表达式中的一种，表9.5列出了Spring Security支持的所有SpEL表达式。

表**9.5　Spring Security**通过一些安全性相关的表达式扩展了**Spring**表达式语言

安全表达式	计 算 结 果
`authentication`	用户的认证对象
`denyAll`	结果始终为false
`hasAnyRole(list of roles)`	如果用户被授予了列表中任意的指定角色，结果为true
`hasRole(role)`	如果用户被授予了指定的角色，结果为true
`hasIpAddress(IP Address)`	如果请求来自指定IP的话，结果为true
`isAnonymous()`	如果当前用户为匿名用户，结果为true
`isAuthenticated()`	如果当前用户进行了认证的话，结果为true
`isFullyAuthenticated()`	如果当前用户进行了完整认证的话（不是通过Remember-me功能进行的认证），结果为true
`isRememberMe()`	如果当前用户是通过Remember-me自动认证的，结果为true

permitAll	结果始终为true
principal	用户的principal对象

在掌握了Spring Security的SpEL表达式后，我们就能够不再局限于基于用户的权限进行访问限制了。例如，如果你想限制“/spitter/me” URL的访问，不仅需要ROLE_SPITTER，还需要来自指定的IP地址，那么我们可以按照如下的方式调用access()方法：

```
.antMatchers("/spitter/me")
  .access("hasRole('ROLE_SPITTER') and hasIpAddress('192.168.1.2')")
```

我们可以使用SpEL实现各种各样的安全性限制。我敢打赌，你已经在想象基于SpEL所能实现的那些有趣的安全性限制了。

但现在，让我们看一下Spring Security拦截请求的另外一种方式：强制通道的安全性。

9.3.2 强制通道的安全性

使用HTTP提交数据是一件具有风险的事情。如果使用HTTP发送无关紧要的信息，这可能不是什么大问题。但是如果你通过HTTP发送诸如密码和信用卡号这样的敏感信息的话，那你就是在找麻烦了。通过HTTP发送的数据没有经过加密，黑客就有机会拦截请求并且能够看到他们想看的数据。这就是为什么敏感信息要通过HTTPS来加密发送的原因。

使用HTTPS似乎很简单。你要做的事情只是在URL中的HTTP后加上一个字母“s”就可以了。是这样吗？

这是真的，但这是把使用HTTPS通道的责任放在了错误的地方。通过添加“s”我们就能很容易地实现页面的安全性，但是忘记添加“s”同样也是很容易出现的。如果我们的应用中有多个链接需要HTTPS，估计在其中的一两个上忘记添加“s”的概率还是很高的。

另一方面，你可能还会在原本并不需要HTTPS的地方，误用HTTPS。

传递到configure()方法中的HttpSecurity对象，除了具有authorizeRequests()方法以外，还有一

个requiresChannel()方法，借助这个方法能够为各种URL模式声明所要求的通道。

作为示例，可以参考Spittr应用的注册表单。尽管Spittr应用不需要信用卡号、社会保障号或其他特别敏感的信息，但用户有可能仍然希望信息是私密的。为了保证注册表单的数据通过HTTPS传送，我们可以在配置中添加requiresChannel()方法，如下所示：

程序清单9.5　requiresChannel()方法会为选定的URL强制使用HTTPS

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/spitter/me").hasRole("SPITTER")
      .antMatchers(HttpMethod.POST, "/spittles").hasRole("SPITTER")
      .anyRequest().permitAll();
    .and()
    .requiresChannel()
      .antMatchers("/spitter/form").requiresSecure();    <—— 需要 HTTPS
}
```

不论何时，只要是对"/spitter/form"的请求，Spring Security都视为需要安全通道（通过调用requiresChannel()确定的）并自动将请求重定向到HTTPS上。

与之相反，有些页面并不需要通过HTTPS传送。例如，首页不包含任何敏感信息，因此并不需要通过HTTPS传送。我们可以使用requiresInsecure()代替requiresSecure()方法，将首页声明为始终通过HTTP传送：

```
.antMatchers("/").requiresInecure();
```

如果通过HTTPS发送了对"/"的请求，Spring Security将会把请求重定向到不安全的HTTP通道上。

在强制要求通道时，路径的选取方案与authorizeRequests()是相同的。在程序清单9.5中，使用了antMatches()，但我们也可以使用regexMatchers()方法，通过正则表达式选取路径模式。

9.3.3　防止跨站请求伪造

我们可以回忆一下，当一个POST请求提交到"/spittles"上时，SpittleController将会为用户创建一个新的Spittle对象。但是，如果这个POST请求来源于其他站点的话，会怎么样呢？如果在

其他站点提交如下表单，这个POST请求会造成什么样的结果呢？

```
<form method="POST" action="http://www.spittr.com/spittles">
  <input type="hidden" name="message" value="I'm stupid!" />
  <input type="submit" value="Click here to win a new car!" />
</form>
```

假设你禁不住获得一辆新汽车的诱惑，点击了按钮——那么你将会提交表单到如下地址http://www.spittr.com/spittles。如果你已经登录到了spittr.com，那么这就会广播一条消息，让每个人都知道你做了一件蠢事。

这是跨站请求伪造（cross-site request forgery，CSRF）的一个简单样例。简单来讲，如果一个站点欺骗用户提交请求到其他服务器的话，就会发生CSRF攻击，这可能会带来消极的后果。尽管提交“I'm stupid!”这样的信息到微博站点算不上什么CSRF攻击的最糟糕场景，但是你可以很容易想到更为严重的攻击情景，它可能会对你的银行账号执行难以预期的操作。

从Spring Security 3.2开始，默认就会启用CSRF防护。实际上，除非你采取行为处理CSRF防护或者将这个功能禁用，否则的话，在应用中提交表单时，你可能会遇到问题。

Spring Security通过一个同步token的方式来实现CSRF防护的功能。它将会拦截状态变化的请求（例如，非GET、HEAD、OPTIONS和TRACE的请求）并检查CSRF token。如果请求中不包含CSRF token的话，或者token不能与服务器端的token相匹配，请求将会失败，并抛出CsrfException异常。

这意味着在你的应用中，所有的表单必须在一个“_csrf”域中提交token，而且这个token必须要与服务器端计算并存储的token一致，这样的话当表单提交的时候，才能进行匹配。

好消息是，Spring Security已经简化了将token放到请求的属性中这一任务。如果你使用Thymeleaf作为页面模板的话，只要<form>标签的action属性添加了Thymeleaf命名空间前缀，那么就会自动生成一个“_csrf”隐藏域：

```
<form method="POST" th:action="@{/spittles}">
   ...
</form>
```

如果使用JSP作为页面模板的话，我们要做的事情非常类似：

```
<input type="hidden"
       name="${_csrf.parameterName}"
       value="${_csrf.token}" />
```

更好的功能是，如果使用Spring的表单绑定标签的话，`<sf:form>`标签会自动为我们添加隐藏的CSRF token标签。

处理CSRF的另外一种方式就是根本不去处理它。我们可以在配置中通过调用`csrf().disable()`禁用Spring Security的CSRF防护功能，如下所示：

程序清单**9.6**　我们可以禁用**Spring Security**的**CSRF**防护功能

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    ...
    .csrf()
      .disable();     <— 禁用 CSRF 防护功能
}
```

需要提醒的是，禁用CSRF防护功能通常来讲并不是一个好主意。如果这样做的话，那么应用就会面临CSRF攻击的风险。只有在深思熟虑之后，才能使用程序清单9.6中的配置。

我们已经配置好了用户存储，也配置好了使用Spring Security来拦截请求，那么接下来就该提示用户输入凭证了。

9.4　认证用户

如果你使用程序清单9.1中最简单的Spring Security配置的话，那么就能无偿地得到一个登录页。实际上，在重写`configure(HttpSecurity)`之前，我们都能使用一个简单却功能完备的登录页。但是，一旦重写了`configure(HttpSecurity)`方法，就失去了这个简单的登录页面。

不过，把这个功能找回来也很容易。我们所需要做的就是在`configure(HttpSecurity)`方法中，调用`formLogin()`，如下面的程序清单所示。

请注意，和前面一样，这里调用`add()`方法来将不同的配置指令连接在一起。

如果我们访问应用的“/login”链接或者导航到需要认证的页面，那么

将会在浏览器中展现登录页面。如图9.2所示，在审美上它没有什么令人兴奋的，但是它却能实现所需的功能。

程序清单9.7　**formLogin()**方法启用了基本的登录页功能

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .formLogin()                    <— 启用默认的登录页
     .and()
    .authorizeRequests()
      .antMatchers("/spitter/me").hasRole("SPITTER")
      .antMatchers(HttpMethod.POST, "/spittles").hasRole("SPITTER")
      .anyRequest().permitAll();
    .and()
    .requiresChannel()
      .antMatchers("/spitter/form").requiresSecure();
}
```

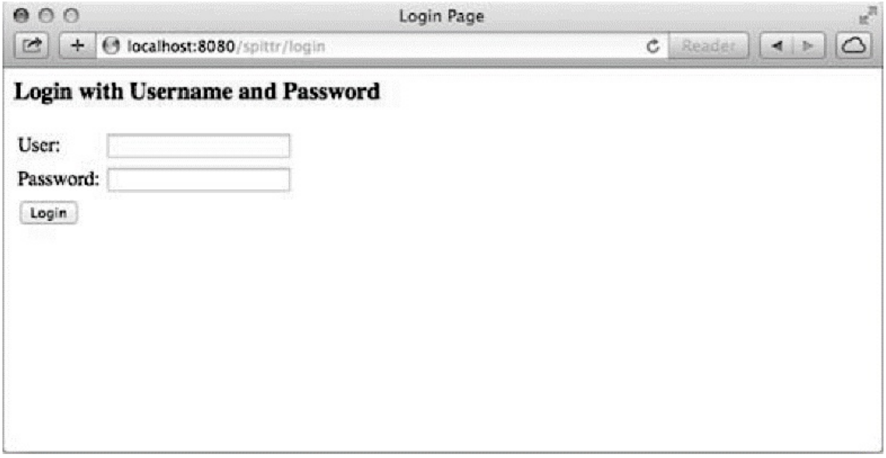

图9.2　默认的登录页在审美上过于简陋，但是功能完备

我敢打赌，你肯定希望在自己的应用程序中能有一个比默认登录页更漂亮的登录页面。如果这个普通的登录页面破坏了我们原本精心设计的漂亮站点，那真的是件很令人遗憾的事情。没问题！接下来，我们就看一下如何为应用添加自定义的登录页面。

9.4.1　添加自定义的登录页

创建自定义登录页的第一步就是了解登录表单中都需要些什么。只需看一下默认登录页面的HTML源码，我们就能了解需要些什么：

```
<html>
<head><title>Login Page</title></head>
<body onload='document.f.username.focus();'>
<h3>Login with Username and Password</h3>
<form name='f' action='/spittr/login' method='POST'>
 <table>
    <tr><td>User:</td><td>
        <input type='text' name='username' value=''></td></tr>
    <tr><td>Password:</td>
        <td><input type='password' name='password'/></td></tr>
    <tr><td colspan='2'>
        <input name="submit" type="submit" value="Login"/></td></tr>
    <input name="_csrf" type="hidden"
          value="6829b1ae-0a14-4920-aac4-5abbd7eeb9ee" />
  </table>
</form>
</body>
</html>
```

需要注意的一个关键点是<form>提交到了什么地方。同时还需要注意username和password输入域，在你的登录页中，需要同样的输入域。最后，假设没有禁用CSRF的话，还需要保证包含了值为CSRF token的“_csrf”输入域。

如下程序清单所展现的Thymeleaf模板提供了一个与Spittr应用风格一致的登录页。

程序清单**9.8**　为**Spittr**应用编写的自定义登录页（以**Thymeleaf**模板的形式）

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Spitter</title>
    <link rel="stylesheet"
          type="text/css"
          th:href="@{/resources/style.css}"></link>
  </head>
  <body onload='document.f.username.focus();'>
    <div id="header" th:include="page :: header"></div>

  <div id="content">
  <form name='f' th:action='@{/login}' method='POST'>      <-- 提交到“/login”
    <table>
    <tr><td>User:</td><td>
        <input type='text' name='username' value='' /></td></tr>
    <tr><td>Password:</td>
        <td><input type='password' name='password'/></td></tr>
    <tr><td colspan='2'>
        <input name="submit" type="submit" value="Login"/></td></tr>
    </table>
   </form>
   </div>
   <div id="footer" th:include="page :: copy"></div>
   </body>
</html>
```

需要注意的是，在Thymeleaf模板中，包含了username和password输入

域，就像默认的登录页一样，它也提交到了相对于上下文的“/login”页面上。因为这是一个Thymeleaf模板，因此隐藏的“_csrf”域将会自动添加到表单中。

9.4.2 启用HTTP Basic认证

对于应用程序的人类用户来说，基于表单的认证是比较理想的。但是在第16章中，将会看到如何将我们Web应用的页面转化为RESTful API。当应用程序的使用者是另外一个应用程序的话，使用表单来提示登录的方式就不太适合了。

HTTP Basic认证（HTTP Basic Authentication）会直接通过HTTP请求本身，对要访问应用程序的用户进行认证。你可能在以前见过HTTP Basic认证。当在Web浏览器中使用时，它将向用户弹出一个简单的模态对话框。

但这只是Web浏览器的显示方式。本质上，这是一个HTTP 401响应，表明必须要在请求中包含一个用户名和密码。在REST客户端向它使用的服务进行认证的场景中，这种方式比较适合。

如果要启用HTTP Basic认证的话，只需在configure()方法所传入的HttpSecurity对象上调用httpBasic()即可。另外，还可以通过调用realmName()方法指定域。如下是在Spring Security中启用HTTP Basic认证的典型配置：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .formLogin()
      .loginPage("/login")
    .and()
    .httpBasic()
      .realmName("Spittr")
    .and()
  ...
}
```

注意，和前面一样，在configure()方法中，通过调用add()方法来将不同的配置指令连接在一起。

在httpBasic()方法中，并没有太多的可配置项，甚至不需要什么额外配置。HTTP Basic认证要么开启要么关闭。所以，与其进一步研究这个话题，还不如看看如何通过Remember-me功能实现用户的自动认证。

9.4.3　启用Remember-me功能

对于应用程序来讲，能够对用户进行认证是非常重要的。但是站在用户的角度来讲，如果应用程序不用每次都提示他们登录是更好的。这就是为什么许多站点提供了Remember-me功能，你只要登录过一次，应用就会记住你，当再次回到应用的时候你就不需要登录了。

Spring Security使得为应用添加Remember-me功能变得非常容易。为了启用这项功能，只需在`configure()`方法所传入的`HttpSecurity`对象上调用`rememberMe()`即可。

```
@Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .formLogin()
        .loginPage("/login")
      .and()
      .rememberMe()
        .tokenValiditySeconds(2419200)
        .key("spittrKey")
  ...
}
```

在这里，我们通过一点特殊的配置就可以启用Remember-me功能。默认情况下，这个功能是通过在cookie中存储一个token完成的，这个token最多两周内有效。但是，在这里，我们指定这个token最多四周内有效（2,419,200秒）。

存储在cookie中的token包含用户名、密码、过期时间和一个私钥——在写入cookie前都进行了MD5哈希。默认情况下，私钥的名为`SpringSecured`，但在这里我们将其设置为`spitterKey`，使它专门用于Spittr应用。

如此简单。既然Remember-me功能已经启用，我们需要有一种方式来让用户表明他们希望应用程序能够记住他们。为了实现这一点，登录请求必须包含一个名为`remember-me`的参数。在登录表单中，增加一个简单复选框就可以完成这件事情：

```
<input id="remember_me" name="remember-me" type="checkbox"/>
<label for="remember_me" class="inline">Remember me</label>
```

在应用中，与登录同等重要的功能就是退出。如果你启用Remember-me功能的话，更是如此，否则的话，用户将永远登录在这个系统中。我们下面将看一下如何添加退　出功能。

9.4.4 退出

其实，按照我们的配置，退出功能已经启用了，不需要再做其他的配置了。我们需要的只是一个使用该功能的链接。

退出功能是通过Servlet容器中的Filter实现的（默认情况下），这个Filter会拦截针对“/logout”的请求。因此，为应用添加退出功能只需添加如下的链接即可（如下以Thymeleaf代码片段的形式进行了展现）：

```
<a th:href="@{/logout}">Logout</a>
```

当用户点击这个链接的时候，会发起对“/logout”的请求，这个请求会被Spring Security的`LogoutFilter`所处理。用户会退出应用，所有的Remember-me token都会被清除掉。在退出完成后，用户浏览器将会重定向到“/login?logout”，从而允许用户进行再次登录。

如果你希望用户被重定向到其他的页面，如应用的首页，那么可以在`configure()`中进行如下的配置：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .formLogin()
      .loginPage("/login")
    .and()
    .logout()
      .logoutSuccessUrl("/")
  ...
}
```

在这里，和前面一样，通过`add()`连接起了对`logout()`的调用。`logout()`提供了配置退出行为的方法。在本例中，调用`logoutSuccessUrl()`表明在退出成功之后，浏览器需要重定向到“/”。

除了`logoutSuccessUrl()`方法以外，你可能还希望重写默认的`LogoutFilter`拦截路径。我们可以通过调用`logoutUrl()`方法实现这一功能：

```
.logout()
  .logoutSuccessUrl("/")
  .logoutUrl("/signout")
```

到目前为止，我们已经看到了如何在发起请求的时候保护Web应用。这假设安全性主要涉及阻止用户访问没有权限的URL。但是，如果我

们能够不给用户显示其无权访问的连接，那么这也是一个很好的思路。接下来，我们将会看一下如何添加视图级别的安全性。

9.5 保护视图

当为浏览器渲染HTML内容时，你可能希望视图中能够反映安全限制和相关的信息。一个简单的样例就是渲染用户的基本信息（比如显示“您已经以……身份登录”）。或者你想根据用户被授予了什么权限，有条件地渲染特定的视图元素。

在第6章，我们看到了在Spring MVC应用中渲染视图的两个最重要的可选方案：JSP和Thymeleaf。不管你使用哪种方案，都有办法在视图上实现安全性。Spring Security本身提供了一个JSP标签库，而Thymeleaf通过特定的方言实现了与Spring Security的集成。

让我们看一下如何将Spring Security用到视图中，就从Spring Security的JSP标签库开始吧。

9.5.1 使用Spring Security的JSP标签库

Spring Security的JSP标签库很小，只包含三个标签，如表9.6所示。

表**9.6 Spring Security**通过**JSP**标签库在视图层上支持安全性

JSP标签	作　用
<security:accesscontrollist>	如果用户通过访问控制列表授予了指定的权限，那么渲染该标签体中的内容
<security:authentication>	渲染当前用户认证对象的详细信息
<security:authorize>	如果用户被授予了特定的权限或者SpEL表达式的计算结果为true，那么渲染该标签体中的内容

为了使用JSP标签库，我们需要在对应的JSP中声明它：

```
<%@ taglib prefix="security"
        uri="http://www.springframework.org/security/tags" %>
```

只要标签库在JSP文件中进行了声明，我们就可以使用它了。让我们看看Spring Security提供的这三个标签是如何工作的。

访问认证信息的细节

借助Spring Security JSP标签库，所能做到的最简单的一件事情就是便利地访问用户的认证信息。例如，对于Web站点来讲，在页面顶部以用户名标示显示“欢迎”或“您好”信息是很常见的。这恰恰是`<security:authentication>`能为我们所做的事情。例如：

```
Hello <security:authentication property="principal.username" />!
```

其中，property用来标示用户认证对象的一个属性。可用的属性取决于用户认证的方式。但是，我们可以依赖几个通用的属性，在不同的认证方式下，它们都是可用的，如表9.7所示。

表**9.7**　使用**<security:authentication> JSP**标签来访问用户的认证详情

认 证 属 性	描　　述
`authorities`	一组用于表示用户所授予权限的GrantedAuthority对象
`Credentials`	用于核实用户的凭证（通常，这会是用户的密码）
`details`	认证的附加信息（IP地址、证件序列号、会话ID等）
`principal`	用户的基本信息对象

在我们的示例中，实际上渲染的是`principal`属性中嵌套的`username`属性。

当像前面示例那样使用时，`<security:authentication>`将在视图中渲染属性的值。但是如果你愿意将其赋值给一个变量，那只需要在`var`属性中指明变量的名字即可。例如，如下展现了如何将其设置给名为`loginId`的属性：

```
<security:authentication property="principal.username"
        var="loginId"/>
```

这个变量默认是定义在页面作用域内的。但是如果你愿意在其他作用域内创建它，例如请求或会话作用域（或者是能够在javax.servlet.jsp.PageContext中获取的其他作用域），那么可以通过scope属性来声明。例如，要在请求作用域内创建这个变量，那可以使用<security:authentication>按照如下的方式来设置：

```
<security:authentication property="principal.username"
        var="loginId" scope="request" />
```

<security:authentication>标签非常有用，但这只是Spring Security JSP标签库功能的基础功能。让我们来看一下如何根据用户的权限来渲染内容。

条件性的渲染内容

有时候视图上的一部分内容需要根据用户被授予了什么权限来确定是否渲染。对于已经登录的用户显示登录表单，或者对还未登录的用户显示个性化的问候信息都是毫无意义的。

Spring Security的<security:authorize>JSP标签能够根据用户被授予的权限有条件地渲染页面的部分内容。例如，在Spittr应用中，对于没有ROLE_SPITTER角色的用户，我们不会为其显示添加新Spitter记录的表单。程序清单9.9展现了如何使用<security:authorize>标签来为具有ROLE_SPITTER角色的用户显示Spitter表单。

程序清单9.9　使用**<security:authorize>**标签基于**SpEL**进行有条件地渲染

```
<sec:authorize access="hasRole('ROLE_SPITTER')">      <!-- 只有在具有 ROLE_SPITTER 权限时 -->
 <s:url value="/spittles" var="spittle_url" />
  <sf:form modelAttribute="spittle"
           action="${spittle_url}">
    <sf:label path="text"><s:message code="label.spittle"
                 text="Enter spittle:"/></sf:label>
    <sf:textarea path="text" rows="2" cols="40" />
        <sf:errors path="text" />

    <br/>
    <div class="spitItSubmitIt">
      <input type="submit" value="Spit it!"
           class="status-btn round-btn disabled" />
    </div>
  </sf:form>
</sec:authorize>
```

access属性被赋值为一个SpEL表达式，这个表达式的值将确定

<security: authorize>标签主体内的内容是否渲染。这里我们使用了hasRole('ROLE_SPITTER')表达式来确保用户具有ROLE_SPITTER角色。但是，当你设置access属性时，可以任意发挥SpEL的强大威力，包括表9.5所示的Spring Security所提供的表达式。

借助于这些可用的表达式，可以构造出非常有意思的安全性约束。例如，假设应用中有一些管理功能只能对用户名为habuma的用户可用。也许你会像这样使用isAuthenticated()和principal表达式:

```
<security:authorize
   access="isAuthenticated() and principal.username=='habuma'">
  <a href="/admin">Administration</a>
</security:authorize>
```

我相信你能设计出比这个更有意思的表达式,可以尽情发挥你的想象力来构造更多的安全性约束。借助于SpEL，选择其实是无限的。

但是我构造的这个示例还有一件事让人很困惑。尽管我想限制管理功能只能给habuma用户，但使用JSP标签表达式并不见得理想。确实，它能在视图上阻止链接的渲染。但是没有什么可以阻止别人在浏览器的地址栏手动输入“/admin”这个URL。

根据我们在本章前面所学，这是一个很容易解决的问题。在安全配置中，添加一个对antMatchers()方法的调用将会严格限制对“/admin”这个URL的访问。

```
.antMatchers("/admin")
  .access("isAuthenticated() and principal.username=='habuma'");
```

现在，管理功能已经被锁定了。URL地址得到了保护，并且到这个URL的链接在用户没有授权使用的情况下不会显示。但是为了做到这一点，我们需要在两个地方声明SpEL表达式——在安全配置中以及在<security:authorize>标签的access属性中。有没有办法消除这种重复性，并且还要确保只有规则条件满足的情况下才渲染管理功能的链接呢？

这是<security:authorize>的url属性所要做的事情。它不像access属性那样明确声明安全性限制，url属性对一个给定的URL模式会间接引用其安全性约束。鉴于我们已经在Spring Security配置中为“/admin”声明了安全性约束，所以我们可以这样使用url属性：

```
<security:authorize url="/admin">
  <spring:url value="/admin" var="admin_url" />
  <br/><a href="${admin_url}">Admin</a>
</security:authorize>
```

因为只有基本信息中用户名为“habuma”的已认证用户才能访问“/admin” URL，所以只有满足以上条件，`<security:authorize>`标签主体中的内容才会被渲染。我们只在一个地方配置了表达式（安全配置中），但是在两个地方进行了应用。

Spring Security的JSP标签库非常便利，尤其是只给满足条件的用户渲染特定的视图元素时更是如此。如果我们选择Thymeleaf而不是JSP作为视图方案的话，我们其实还能延续这种好运气。我们已经看到Thymeleaf的Spring方言能够自动为表单添加隐藏的CSRF token，现在我们看一下Thymeleaf如何支持Spring Security。

9.5.2 使用Thymeleaf的Spring Security方言

与Spring Security的JSP标签库类似，Thymeleaf的安全方言提供了条件化渲染和显示认证细节的能力。表9.8列出了安全方言所提供的属性。

表9.8 **Thymeleaf**的安全方言提供了与**Spring Security**标签库相对应的属性

属性	作用
`sec:authentication`	渲染认证对象的属性。类似于Spring Security的`<sec:authentication/>`JSP标签
`sec:authorize`	基于表达式的计算结果，条件性的渲染内容。类似于Spring Security的`<sec:authorize/>`JSP标签
`sec:authorize-acl`	基于表达式的计算结果，条件性的渲染内容。类似于Spring Security的`<sec:accesscontrollist/>` JSP标签
`sec:authorize-expr`	sec:authorize属性的别名
`sec:authorize-url`	基于给定URL路径相关的安全规则，条件性的渲染内容。类似于Spring Security的`<sec:authorize/>` JSP标签使用url属性时

	的场景

为了使用安全方言，我们需要确保Thymeleaf Extras Spring Security已经位于应用的类路径下。然后，还需要在配置中使用SpringTemplateEngine来注册SpringSecurity Dialect。程序清单9.10所展现的@Bean方法声明了SpringTemplateEngine bean，其中就包含了SpringSecurityDialect。

程序清单9.10　注册Thymeleaf的Spring Security安全方言

```
@Bean
public SpringTemplateEngine templateEngine(
                              TemplateResolver templateResolver) {
  SpringTemplateEngine templateEngine = new SpringTemplateEngine();
  templateEngine.setTemplateResolver(templateResolver);
  templateEngine.addDialect(new SpringSecurityDialect());   <-- 注册安全方言
   return templateEngine;
}
```

安全方言注册完成之后，我们就可以在Thymeleaf模板中使用它的属性了。首先，需要在使用这些属性的模板中声明安全命名空间：

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:sec=
          "http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
  ...
</html>
```

在这里，标准的Thymeleaf方法依旧与之前一样，使用th前缀，安全方言则设置为使用sec前缀。

这样我们就能在任意合适的地方使用Thymeleaf属性了。比如，假设我们想要为认证用户渲染“Hello”文本。如下的Thymeleaf模板代码片段就能完成这项任务：

```
<div sec:authorize="isAuthenticated()">
  Hello <span sec:authentication="name">someone</span>
</div>
```

sec:authorize属性会接受一个SpEL表达式。如果表达式的计算结果为true，那么元素的主体内容就会渲染。在本例中，表达式为isAuthenticated()，所以只有用户已经进行了认证，才会渲染<div>标签的主体内容。就这个标签的主体内容部分而言，它的功

能是使用认证对象的name属性提示“Hello”文本。

你可能还记得，在Spring Security中，借助<sec:authorize>JSP标签的url属性能够基于给定URL的权限有条件地渲染内容。在Thymeleaf中，我们可以通过sec:authorize-url属性完成相同的功能。例如，如下Thymeleaf代码片段所实现的功能与之前<sec:authorize> JSP标签和url属性所实现的功能是相同的：

```
<span sec:authorize-url="/admin">
  <br/><a th:href="@{/admin}">Admin</a>
</span>
```

如果用户有权限访问“/admin”的话，那么到管理页面的链接就会渲染，否则的话，这个链接将不会渲染。

9.6 小结

对于许多应用而言，安全性都是非常重要的切面。Spring Security提供了一种简单、灵活且强大的机制来保护我们的应用程序。

借助于一系列Servlet Filter，Spring Security能够控制对Web资源的访问，包括Spring MVC控制器。借助于Spring Security的Java配置模型，我们不必直接处理Filter，能够非常简洁地声明Web安全性功能。

当认证用户时，Spring Security提供了多种选项。我们探讨了如何基于内存用户库、关系型数据库和LDAP目录服务器来配置认证功能。如果这些可选方案无法满足认证需求的话，我们还学习了如何创建和配置自定义的用户服务。

在前面的几章中，我们看到了如何将Spring运用到应用程序的前端。在接下来的章中，我们将会继续深入这个技术栈，学习Spring如何在后端发挥作用，下一章将会首先从Spring的JDBC抽象开始。

第3部分　后端中的Spring

尽管用户看到的内容是由Web应用所提供的页面，但是在这背后，实际的工作是在后端服务器中发生的，在这里会处理和持久化数据。第3部分将会关注Spring如何帮助我们在后端处理数据。

多年以来，关系型数据库一直是企业级应用中的统治者。在第10章“通过Spring和JDBC征服数据库”中，我们将会看到如何使用Spring的JDBC抽象来查询关系型数据库，这要比原生的JDBC简单得多。

如果你不喜欢JDBC风格的话，在第11章“通过对象-关系映射持久化数据”中，将会展现如何与ORM框架进行集成，这些框架包括Hibernate以及其他的Java持久化API（Java Persistence API，JPA）实现。除此之外，还将会看到如何发挥Spring Data JPA的魔力，在运行时自动生成Repository实现。

关系型数据库不一定是所有场景下的最佳选择，因此，第12章“使用NoSQL数据库”将会研究其他的Spring Data项目，它们能够持久化各种非关系型数据库中的数据，包括MongoDB、Neo4j和Redis。

第13章“缓存数据”为上述的持久化章提供了一个缓存层，如果数据已经可用的话，它会避免数据库操作，从而提升应用的性能。

与前端类似，安全性在后端也是一个很重要的方面。在第14章“保护方法应用”中，将会把Spring Security应用于后端，它会拦截方法的调用并确保调用者被授予了适当的权限。

第10章　通过Spring和JDBC征服数据库

本章内容：

- 定义Spring对数据访问的支持
- 配置数据库资源
- 使用Spring的JDBC模版

在掌握了Spring容器的核心知识之后，是时候将它在实际应用中进行使用了。数据持久化是一个非常不错的起点，因为几乎所有的企业级应用程序中都存在这样的需求。我们可能都处理过数据库访问功能，在实际的工作中也发现数据访问有一些不足之处。我们必须初始化数据访问框架、打开连接、处理各种异常和关闭连接。如果上述操作出现任何问题，都有可能损坏或删除珍贵的企业数据。如果你还未曾经历过因未妥善处理数据访问而带来的严重后果，那我要提醒你这绝对不是什么好事情。

做事要追求尽善尽美，所以我们选择了Spring。Spring自带了一组数据访问框架，集成了多种数据访问技术。不管你是直接通过JDBC还是像Hibernate这样的对象关系映射（object-relational mapping，ORM）框架实现数据持久化，Spring都能够帮你消除持久化代码中那些单调枯燥的数据访问逻辑。我们可以依赖Spring来处理底层的数据访问，这样就可以专注于应用程序中数据的管理了。

当开发Spittr应用的持久层的时候，会面临多种选择，我们可以使用JDBC、Hibernate、Java持久化API（Java Persistence API，JPA）或者其他任意的持久化框架。你可能还会考虑使用最近很流行的NoSQL数据库（其实我更喜欢将其称为无模式数据库）。

幸好，不管你选择哪种持久化方式，Spring都能够提供支持。在本章，我们主要关注于Spring对JDBC的支持。但首先，我们来熟悉一下Spring的持久化哲学，从而为后面打好基础。

10.1　Spring的数据访问哲学

从前面的几章可以看出，Spring的目标之一就是允许我们在开发应用程序时，能够遵循面向对象（OO）原则中的“针对接口编程”。Spring对数据访问的支持也不例外。

像很多应用程序一样，Spittr应用需要从某种类型的数据库中读取和写入数据。为了避免持久化的逻辑分散到应用的各个组件中，最好将数据访问的功能放到一个或多个专注于此项任务的组件中。这样的组件通常称为数据访问对象（data access object，DAO）或Repository。

为了避免应用与特定的数据访问策略耦合在一起，编写良好的Repository应该以接口的方式暴露功能。图10.1展现了设计数据访问层的合理方式。

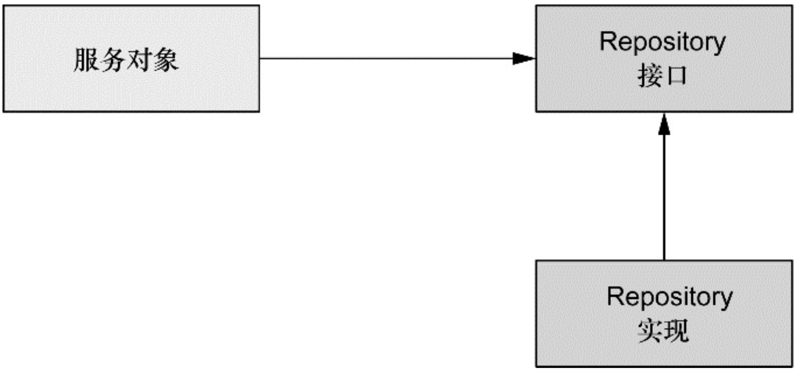

图10.1　服务对象本身并不会处理数据访问，而是将数据访问委托给Repository。Repository接口确保其与服务对象的松耦合

如图所示，服务对象通过接口来访问Repository。这样做会有几个好处。第一，它使得服务对象易于测试，因为它们不再与特定的数据访问实现绑定在一起。实际上，你可以为这些数据访问接口创建mock实现，这样无需连接数据库就能测试服务对象，而且会显著提升单元测试的效率并排除因数据不一致所造成的测试失败。

此外，数据访问层是以持久化技术无关的方式来进行访问的。持久化方式的选择独立于Repository，同时只有数据访问相关的方法才通过接口进行暴露。这可以实现灵活的设计，并且切换持久化框架对应用程序其他部分所带来的影响最小。如果将数据访问层的实现细节渗透到应用程序的其他部分中，那么整个应用程序将与数据访问层耦合在一起，从而导致僵化的设计。

接口与Spring：如果在阅读了上面几段文字之后，你能感受到我倾向于将持久层隐藏在接口之后，那很高兴我的目的达到了。我相信接口是实现松耦合代码的关键，并且应将其用于应用程序的各个层，而不

仅仅是持久化层。还要说明一点，尽管Spring鼓励使用接口，但这并不是强制的——你可以使用Spring将bean（DAO或其他类型）直接装配到另一个bean的某个属性中，而不需要一定通过接口注入。

为了将数据访问层与应用程序的其他部分隔离开来，Spring采用的方式之一就是提供统一的异常体系，这个异常体系用在了它支持的所有持久化方案中。

10.1.1　了解Spring的数据访问异常体系

这里有一个关于跳伞运动员的经典笑话，这个运动员被风吹离正常路线后降落在树上并高高地挂在那里。后来，有人路过，跳伞运动员就问他自己在什么地方。过路人回答说：“你在离地大约20尺的空中。”跳伞运动员说：“你一定是个软件分析师。”过路人回应说“你说对了。你是怎么知道的呢？”“因为你跟我说的话百分百正确，但丝毫用处都没有。”

这个故事已经听过很多遍了，每次过路人的职业或国籍都会有所不同。但是这个故事使我想起了JDBC中的`SQLException`。如果你曾经编写过JDBC代码（不使用Spring），你肯定会意识到如果不强制捕获`SQLException`的话，几乎无法使用JDBC做任何事情。`SQLException`表示在尝试访问数据库的时出现了问题，但是这个异常却没有告诉你哪里出错了以及如何进行处理。

可能导致抛出`SQLException`的常见问题包括：

- 应用程序无法连接数据库；
- 要执行的查询存在语法错误；
- 查询中所使用的表和/或列不存在；
- 试图插入或更新的数据违反了数据库约束。

`SQLException`的问题在于捕获到它的时候该如何处理。事实上，能够触发`SQLException`的问题通常是不能在catch代码块中解决的。大多数抛出`SQLException`的情况表明发生了致命性错误。如果应用程序不能连接到数据库，这通常意味着应用不能继续使用了。类似地，如果查询时出现了错误，那在运行时基本上也是无能为力。

如果无法从`SQLException`中恢复，那为什么我们还要强制捕获它呢？

即使对某些`SQLException`有处理方案，我们还是要捕获

SQLException并查看其属性才能获知问题根源的更多信息。这是因为SQLException被视为处理数据访问所有问题的通用异常。对于所有的数据访问问题都会抛出SQLException，而不是对每种可能的问题都会有不同的异常类型。

一些持久化框架提供了相对丰富的异常体系。例如，Hibernate提供了二十个左右的异常，分别对应于特定的数据访问问题。这样就可以针对想处理的异常编写catch代码块。

即便如此，Hibernate的异常是其本身所特有的。正如前面所言，我们想将特定的持久化机制独立于数据访问层。如果抛出了Hibernate所特有的异常，那我们对Hibernate的使用将会渗透到应用程序的其他部分。如果不这样做的话，我们就得捕获持久化平台的异常，然后将其作为平台无关的异常再次抛出。

一方面，JDBC的异常体系过于简单了——实际上，它算不上一个体系。另一方面，Hibernate的异常体系是其本身所独有的。我们需要的数据访问异常要具有描述性而且又与特定的持久化框架无关。

Spring所提供的平台无关的持久化异常

Spring JDBC提供的数据访问异常体系解决了以上的两个问题。不同于JDBC，Spring提供了多个数据访问异常，分别描述了它们抛出时所对应的问题。表10.1对比了Spring的部分数据访问异常以及JDBC所提供的异常。

从表中可以看出，Spring为读取和写入数据库的几乎所有错误都提供了异常。Spring的数据访问异常要比表10.1所列的还要多。（在此没有列出所有的异常，因为我不想让JDBC显得太寒酸。）

表**10.1**　**JDBC**的异常体系与**Spring**的数据访问异常

JDBC的异常	**Spring**的数据访问异常
BatchUpdateException DataTruncation SQLException SQLWarning	BadSqlGrammarException CannotAcquireLockException CannotSerializeTransactionException CannotGetJdbcConnectionException CleanupFailureDataAccessException ConcurrencyFailureException DataAccessException DataAccessResourceFailureException DataIntegrityViolationException DataRetrievalFailureException DataSourceLookupApiUsageException DeadlockLoserDataAccessException

SQLWarning	DuplicateKeyException EmptyResultDataAccessException IncorrectResultSizeDataAccessException IncorrectUpdateSemanticsDataAccessException InvalidDataAccessApiUsageException InvalidDataAccessResourceUsageException InvalidResultSetAccessException JdbcUpdateAffectedIncorrectNumberOfRowsException LbRetrievalFailureException
BatchUpdateException DataTruncation SQLException SQLWarning	NonTransientDataAccessResourceException OptimisticLockingFailureException PermissionDeniedDataAccessException PessimisticLockingFailureException QueryTimeoutException RecoverableDataAccessException SQLWarningException SqlXmlFeatureNotImplementedException TransientDataAccessException TransientDataAccessResourceException TypeMismatchDataAccessException UncategorizedDataAccessException UncategorizedSQLException

尽管Spring的异常体系比JDBC简单的`SQLException`丰富得多，但它并没有与特定的持久化方式相关联。这意味着我们可以使用Spring抛出一致的异常，而不用关心所选择的持久化方案。这有助于我们将所选择持久化机制与数据访问层隔离开来。

看！不用写**catch**代码块

表10.1中没有体现出来的一点就是这些异常都继承自`DataAccessException`。`DataAccessException`的特殊之处在于它是一个非检查型异常。换句话说，没有必要捕获Spring所抛出的数据访问异常（当然，如果你想捕获的话也是完全可以的）。

`DataAccessException`只是Sping处理检查型异常和非检查型异常哲学的一个范例。Spring认为触发异常的很多问题是不能在`catch`代码块中修复的。Spring使用了非检查型异常，而不是强制开发人员编写`catch`代码块（里面经常是空的）。这把是否要捕获异常的权力留给了开发人员。

为了利用Spring的数据访问异常，我们必须使用Spring所支持的数据访问模板。让我们看一下Spring的模板是如何简化数据访问的。

10.1.2 数据访问模板化

如果以前有搭乘飞机旅行的经历，你肯定会觉得旅行中很重要的一件

事就是将行李从一个地方搬运到另一个地方。这个过程包含多个步骤。当你到达机场时，第一站是到柜台办理行李托运。然后保安人员对其进行安检以确保安全。之后行李将通过行李车转送到飞机上。如果你需要中途转机，行李也要进行中转。当你到达目的地的时候，行李需要从飞机上取下来并放到传送带上。最后，你到行李认领区将其取回。

尽管在这个过程中包含多个步骤，但是涉及到旅客的只有几个。承运人负责推动整个流程。你只会在必要的时候进行参与，其余的过程不必关心。这反映了一个强大的设计模式：模板方法模式。

模板方法定义过程的主要框架。在我们的示例中，整个过程是将行李从出发地运送到目的地。过程本身是固定不变的。处理行李过程中的每个事件都会以同样的方式进行：托运检查、运送到飞机上等等。在这个过程中的某些步骤是固定的——这些步骤每次都是一样的。比如当飞机到达目的地后，所有的行李被取下来并通过传送带运到取行李处。

在某些特定的步骤上，处理过程会将其工作委派给子类来完成一些特定实现的细节。这是过程中变化的部分。例如，处理行李是从乘客在柜台托运行李开始的。这部分的处理往往是在最开始的时候进行，所以它在处理过程中的顺序是固定的。由于每位乘客的行李登记都不一样，所以这个过程的实现是由旅客决定的。按照软件方面的术语来讲，模板方法将过程中与特定实现相关的部分委托给接口，而这个接口的不同实现定义了过程中的具体行为。

这也是Spring在数据访问中所使用的模式。不管我们使用什么样的技术，都需要一些特定的数据访问步骤。例如，我们都需要获取一个到数据存储的连接并在处理完成后释放资源。这都是在数据访问处理过程中的固定步骤，但是每种数据访问方法又会有些不同，我们会查询不同的对象或以不同的方式更新数据，这都是数据访问过程中变化的部分。

Spring将数据访问过程中固定的和可变的部分明确划分为两个不同的类：模板（template）和回调（callback）。模板管理过程中固定的部分，而回调处理自定义的数据访问代码。图10.2展现了这两个类的职责。

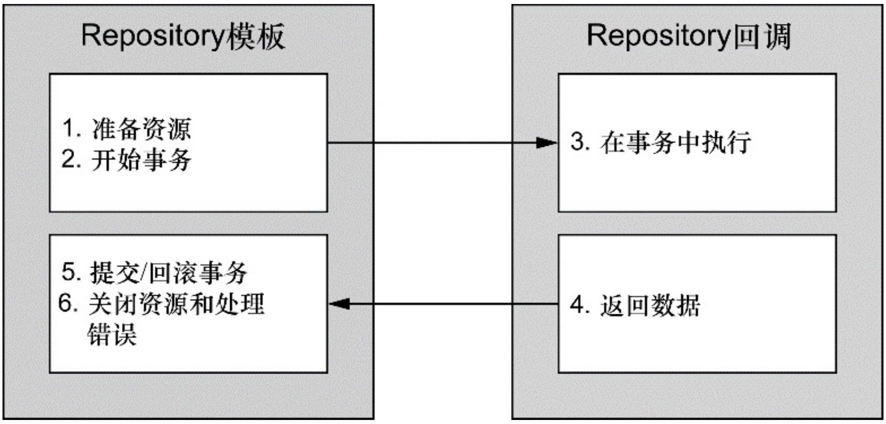

图10.2　Spring的数据访问模板类负责通用的数据访问功能。对于应用程序特定的任务，则会调用自定义的回调对象

如图所示，Spring的模板类处理数据访问的固定部分——事务控制、管理资源以及处理异常。同时，应用程序相关的数据访问——语句、绑定参数以及整理结果集——在回调的实现中处理。事实证明，这是一个优雅的架构，因为你只需关心自己的数据访问逻辑即可。

针对不同的持久化平台，Spring提供了多个可选的模板。如果直接使用JDBC，那你可以选择`JdbcTemplate`。如果你希望使用对象关系映射框架，那`HibernateTemplate`或`JpaTemplate`可能会更适合你。表10.2列出了Spring所提供的所有数据访问模板及其用途。

表**10.2**　**Spring**提供的数据访问模板，分别适用于不同的持久化机制

模板类（**org.springframework.***）	用　　途
`jca.cci.core.CciTemplate`	JCA CCI连接
`jdbc.core.JdbcTemplate`	JDBC连接
`jdbc.core.namedparam.NamedParameterJdbcTemplate`	支持命名参数的JDBC连接
`jdbc.core.simple.SimpleJdbcTemplate`	通过Java 5简化后的JDBC连接（Spring 3.1中已经废弃）
`orm.hibernate3.HibernateTemplate`	Hibernate 3.x以上的Session

orm.ibatis.SqlMapClientTemplate	iBATIS SqlMap客户端
orm.jdo.JdoTemplate	Java数据对象（Java Data Object）实现
orm.jpa.JpaTemplate	Java持久化API的实体管理器

Spring为多种持久化框架提供了支持，这里没有那么多的篇幅在本章对其进行一一介绍。因此，我会关注于我认为最为实用的持久化方案，这也是读者最可能用到的。

在本章中，我们将会从基础的JDBC访问开始，因为这是从数据库中读取和写入数据的最基本方式。在第11章中，我们将会了解Hibernate和JPA，这是最流行的基于POJO的ORM方案。我们会在第12章结束Spring持久化的话题，在这一章中，将会看到Spring Data项目是如何让Spring支持无模式数据的。

但首先要说明的是Spring所支持的大多数持久化功能都依赖于数据源。因此，在声明模板和Repository之前，我们需要在Spring中配置一个数据源用来连接数据库。

10.2 配置数据源

无论选择Spring的哪种数据访问方式，你都需要配置一个数据源的引用。Spring提供了在Spring上下文中配置数据源bean的多种方式，包括：

- 通过JDBC驱动程序定义的数据源；
- 通过JNDI查找的数据源；
- 连接池的数据源。

对于即将发布到生产环境中的应用程序，我建议使用从连接池获取连接的数据源。如果可能的话，我倾向于通过应用服务器的JNDI来获取数据源。请记住这一点，让我们首先看一下如何配置Spring从JNDI中获取数据源。

10.2.1 使用JNDI数据源

Spring应用程序经常部署在Java EE应用服务器中，如WebSphere、JBoss或甚至像Tomcat这样的Web容器中。这些服务器允许你配置通过JNDI获取数据源。这种配置的好处在于数据源完全可以在应用程序之外进行管理，这样应用程序只需在访问数据库的时候查找数据源就可以了。另外，在应用服务器中管理的数据源通常以池的方式组织，从而具备更好的性能，并且还支持系统管理员对其进行热切换。

利用Spring，我们可以像使用Spring bean那样配置JNDI中数据源的引用并将其装配到需要的类中。位于jee命名空间下的<jee:jndi-lookup>元素可以用于检索JNDI中的任何对象（包括数据源）并将其作为Spring的bean。例如，如果应用程序的数据源配置在JNDI中，我们可以使用<jee:jndi-lookup>元素将其装配到Spring中，如下所示：

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
resource-ref="true" />
```

其中jndi-name属性用于指定JNDI中资源的名称。如果只设置了jndi-name属性，那么就会根据指定的名称查找数据源。但是，如果应用程序运行在Java应用服务器中，你需要将resource-ref属性设置为true，这样给定的jndi-name将会自动添加“java:comp/env/”前缀。

如果想使用Java配置的话，那我们可以借助JndiObjectFactoryBean从JNDI中查找DataSource：

```
@Bean
public JndiObjectFactoryBean dataSource() {
  JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();
  jndiObjectFB.setJndiName("jdbc/SpittrDS");
  jndiObjectFB.setResourceRef(true);
  jndiObjectFB.setProxyInterface(javax.sql.DataSource.class);
  return jndiObjectFB;
}
```

显然，通过Java配置获取JNDI bean要更为复杂。大多数情况下，Java配置要比XML配置简单，但是这一次我们需要写更多的Java代码。但是，很容易就能够看出Java代码中与XML相对应的配置，Java配置的内容其实也不算多。

10.2.2 使用数据源连接池

如果你不能从JNDI中查找数据源，那么下一个选择就是直接在Spring中配置数据源连接池。尽管Spring并没有提供数据源连接池实现，但

是我们有多项可用的方案，包括如下开源的实现：

- Apache Commons DBCP (http://jakarta.apache.org/commons/dbcp)；
- c3p0 (http://sourceforge.net/projects/c3p0/) ；
- BoneCP (http://jolbox.com/) 。

这些连接池中的大多数都能配置为Spring的数据源，在一定程度上与Spring自带的DriverManagerDataSource或SingleConnectionDataSource很类似（我们稍后会对其进行介绍）。例如，如下就是配置DBCP BasicDataSource的方式：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  p:driverClassName="org.h2.Driver"
  p:url="jdbc:h2:tcp://localhost/~/spitter"
  p:username="sa"
  p:password=""
  p:initialSize="5"
  p:maxActive="10" />
```

如果你喜欢Java配置的话，连接池形式的DataSourcebean可以声明如下：

```
@Bean
public BasicDataSource dataSource() {

  BasicDataSource ds = new BasicDataSource();
  ds.setDriverClassName("org.h2.Driver");
  ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
  ds.setUsername("sa");
  ds.setPassword("");
  ds.setInitialSize(5);
  ds.setMaxActive(10);
  return ds;
}
```

前四个属性是配置BasicDataSource所必需的。属性driverClassName指定了JDBC驱动类的全限定类名。在这里我们配置的是H2数据库的数据源。属性url用于设置数据库的JDBC URL。最后，username和password用于在连接数据库时进行认证。

以上四个基本属性定义了BasicDataSource的连接信息。除此以外，还有多个配置数据源连接池的属性。表10.3列出了DBCP BasicDataSource最有用的一些池配置属性：

表**10.3　BasicDataSource**的池配置属性

池配置属性	所指定的内容

initialSize	池启动时创建的连接数量
maxActive	同一时间可从池中分配的最多连接数。如果设置为0，表示无限制
maxIdle	池里不会被释放的最多空闲连接数。如果设置为0，表示无限制
maxOpenPreparedStatements	在同一时间能够从语句池中分配的预处理语句（prepared statement）的最大数量。如果设置为0，表示无限制
maxWait	在抛出异常之前，池等待连接回收的最大时间（当没有可用连接时）。如果设置为-1，表示无限等待
minEvictableIdleTimeMillis	连接在池中保持空闲而不被回收的最大时间
minIdle	在不创建新连接的情况下，池中保持空闲的最小连接数
poolPreparedStatements	是否对预处理语句（prepared statement）进行池管理（布尔值）

在我们的示例中，连接池启动时会创建5个连接；当需要的时候，允许BasicDataSource创建新的连接，但最大活跃连接数为10。

10.2.3 基于JDBC驱动的数据源

在Spring中，通过JDBC驱动定义数据源是最简单的配置方式。Spring提供了三个这样的数据源类（均位于org.springframework.jdbc.datasource包中）供选择：

- DriverManagerDataSource：在每个连接请求时都会返回一个新建的连接。与DBCP的BasicDataSource不同，由DriverManagerDataSource提供的连接并没有进行池化管理；
- SimpleDriverDataSource：

与DriverManagerDataSource的工作方式类似，但是它直接使用JDBC驱动，来解决在特定环境下的类加载问题，这样的环境包括OSGi容器；

- SingleConnectionDataSource：在每个连接请求时都会返回同一个的连接。尽管SingleConnectionDataSource不是严格意义上的连接池数据源，但是你可以将其视为只有一个连接的池。

以上这些数据源的配置与DBCPBasicDataSource的配置类似。例如，如下就是配置DriverManagerDataSource的方法：

```
@Bean
public DataSource dataSource() {
  DriverManagerDataSource ds = new DriverManagerDataSource();
  ds.setDriverClassName("org.h2.Driver");
  ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
  ds.setUsername("sa");
  ds.setPassword("");
  return ds;
}
```

如果使用XML的话，DriverManagerDataSource可以按照如下的方式配置：

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource"
  p:driverClassName="org.h2.Driver"
  p:url="jdbc:h2:tcp://localhost/~/spitter"
  p:username="sa"
  p:password="" />
```

与具备池功能的数据源相比，唯一的区别在于这些数据源bean都没有提供连接池功能，所以没有可配置的池相关的属性。

尽管这些数据源对于小应用或开发环境来说是不错的，但是要将其用于生产环境，你还是需要慎重考虑。因为SingleConnectionDataSource有且只有一个数据库连接，所以不适合用于多线程的应用程序，最好只在测试的时候使用。而DriverManagerDataSource和SimpleDriverDataSource尽管支持多线程，但是在每次请求连接的时候都会创建新连接，这是以性能为代价的。鉴于以上的这些限制，我强烈建议应该使用数据源连接池。

10.2.4 使用嵌入式的数据源

除此之外，还有一个数据源是我想对读者介绍的：嵌入式数据库（embedded database）。嵌入式数据库作为应用的一部分运行，而不

是应用连接的独立数据库服务器。尽管在生产环境的设置中，它并没有太大的用处，但是对于开发和测试来讲，嵌入式数据库都是很好的可选方案。这是因为每次重启应用或运行测试的时候，都能够重新填充测试数据。

Spring的jdbc命名空间能够简化嵌入式数据库的配置。例如，如下的程序清单展现了如何使用jdbc命名空间来配置嵌入式的H2数据库，它会预先加载一组测试数据。

程序清单**10.1**　使用**jdbc**命名空间配置嵌入式数据库

```
<?xml version="1.0" encoding="UTF-
     8"?> <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
...
  <jdbc:embedded-
     database id="dataSource" type="H2">     <jdbc:script location="com/habum
     a/spitter/db/jdbc/schema.sql"/>     <jdbc:script location="com/habuma/sp
     itter/db/jdbc/test-data.sql"/>   </jdbc:embedded-database>
...
</beans>
```

我们将<jdbc:embedded-database>的type属性设置为H2，表明嵌入式数据库应该是H2数据库（要确保H2位于应用的类路径下）。另外，我们还可以将type设置为DERBY，以使用嵌入式的Apache Derby数据库。

在<jdbc:embedded-database>中，我们可以不配置也可以配置多个<jdbc:script>元素来搭建数据库。程序清单10.1中包含了两个<jdbc:script>元素：第一个引用了schema.sql，它包含了在数据库中创建表的SQL；第二个引用了test-data.sql，用来将测试数据填充到数据库中。

除了搭建嵌入式数据库以外，<jdbc:embedded-database>元素还会暴露一个数据源，我们可以像使用其他的数据源那样来使用它。在这里，id属性被设置成了dataSource，这也是所暴露数据源的bean ID。因此，当我们需要javax.sql.DataSource的时候，就可以注入dataSource bean。

如果使用Java来配置嵌入式数据库时，不会像jdbc命名空间那么简便，我们可以使用EmbeddedDatabaseBuilder来构建

DataSource:

```
@Bean
public DataSource dataSource() {
  return new EmbeddedDatabaseBuilder()
      .setType(EmbeddedDatabaseType.H2)
      .addScript("classpath:schema.sql")
      .addScript("classpath:test-data.sql")
      .build();
}
```

可以看到，setType()方法等同于<jdbc:embedded-database>元素中的type属性，此外，我们这里用addScript()代替<jdbc:script>元素来指定初始化SQL。

10.2.5 使用profile选择数据源

我们已经看到了多种在Spring中配置数据源的方法，我相信你已经找到了一两种适合你的应用程序的配置方式。实际上，我们很可能面临这样一种需求，那就是在某种环境下需要其中一种数据源，而在另外的环境中需要不同的数据源。

例如，对于开发期来说，<jdbc:embedded-database>元素是很合适的，而在QA环境中，你可能希望使用DBCP的BasicDataSource，在生产部署环境下，可能需要使用<jee:jndi-lookup>。

我们在第3章所讨论的Spring的bean profile特性恰好用在这里，所需要做的就是将每个数据源配置在不同的profile中，如下所示：

程序清单**10.2** 借助**Spring**的**profile**特性能够在运行时选择数据源

```
package com.habuma.spittr.config;
import org.apache.commons.dbcp.BasicDataSource;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration
public class DataSourceConfiguration {
                                                    开发数据源
  @Profile("development")                       <—
  @Bean
  public DataSource embeddedDataSource() {
    return new EmbeddedDatabaseBuilder()
               .setType(EmbeddedDatabaseType.H2)
               .addScript("classpath:schema.sql")
               .addScript("classpath:test-data.sql")
               .build();
  }

  @Profile("qa")                          <— QA 数据源
  @Bean
  public DataSource Data() {
    BasicDataSource ds = new BasicDataSource();
    ds.setDriverClassName("org.h2.Driver");
    ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
    ds.setUsername("sa");
    ds.setPassword("");
    ds.setInitialSize(5);
    ds.setMaxActive(10);
    return ds;
  }

  @Profile("production")                  <— 生产环境的数据源
  @Bean
  public DataSource dataSource() {
    JndiObjectFactoryBean jndiObjectFactoryBean
                                    = new JndiObjectFactoryBean();
    jndiObjectFactoryBean.setJndiName("jdbc/SpittrDS");
    jndiObjectFactoryBean.setResourceRef(true);
    jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
    return (DataSource) jndiObjectFactoryBean.getObject();
  }
}
```

通过使用profile功能，会在运行时选择数据源，这取决于哪一个profile处于激活状态。如程序清单10.2配置所示，当且仅当`development`profile处于激活状态时，会创建嵌入式数据库，当且仅当`qa` profile处于激活状态时，会创建DBCP `BasicDataSource`，当且仅当`production`profile处于激活状态时，会从JNDI获取数据源。

为了内容的完整性，如下的程序清单展现了如何使用Spring XML代替Java配置，实现相同的profile配置。

程序清单**10.3**　借助**XML**配置，基于**profile**选择数据源

```
<?xml version="1.0" encoding="UTF-
     8"?> <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
    http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
  <beans profile="development">             <!-- 开发数据源 -->
  <jdbc:embedded-
   database id="dataSource" type="H2">       <jdbc:script location="com/hab
   uma/spitter/db/jdbc/schema.sql"/>       <jdbc:script location="com/habum
   a/spitter/db/jdbc/test-data.sql"/>     </jdbc:embedded-
   database>   </beans>
 <beans profile="qa">                          <!-- QA 数据源 -->
   <bean id="dataSource"
         class="org.apache.commons.dbcp.BasicDataSource"
     p:driverClassName="org.h2.Driver"
     p:url="jdbc:h2:tcp://localhost/~/spitter"
     p:username="sa"
     p:password=""
     p:initialSize="5"
     p:maxActive="10" />   </beans>
 <beans profile="production">                  <!-- 生产环境的数据源 -->
   <jee:jndi-lookup id="dataSource"
                    jndi-name="/jdbc/SpitterDS"
                    resource-ref="true" />   </beans>
</beans>
```

现在我们已经通过数据源建立了与数据库的连接，接下来要实际访问数据库了。就像我在前面所提到的，Spring为我们提供了多种使用数据库的方式包括JDBC、Hibernate以及Java持久化API（Java Persistence API，JPA）。在下一节，我们将会看到如何使用Spring对JDBC的支持为应用程序构建持久层。如果你喜欢使用Hibernate或JPA，那可以直接跳到下一章。

10.3 在Spring中使用JDBC

持久化技术有很多种，而Hibernate、iBATIS和JPA只是其中的几种而已。尽管如此，还是有很多的应用程序使用最古老的方式将Java对象保存到数据库中：他们自食其力。不，等等，这是他们挣钱的途径。这种久经考验并证明行之有效的持久化方法就是古老的JDBC。

为什么不采用它呢？JDBC不要求我们掌握其他框架的查询语言。它是建立在SQL之上的，而SQL本身就是数据访问语言。此外，与其他的技术相比，使用JDBC能够更好地对数据访问的性能进行调优。JDBC允许你使用数据库的所有特性，而这是其他框架不鼓励甚至禁止的。

再者，相对于持久层框架，JDBC能够让我们在更低的层次上处理数据，我们可以完全控制应用程序如何读取和管理数据，包括访问和管理数据库中单独的列。这种细粒度的数据访问方式在很多应用程序中是很方便的。例如在报表应用中，如果将数据组织为对象，而接下来唯一要做的就是将其解包为原始数据，那就没有太大意义了。

但是JDBC也不是十全十美的。虽然JDBC具有强大、灵活和其他一些优点，但也有其不足之处。

10.3.1　应对失控的JDBC代码

如果使用JDBC所提供的直接操作数据库的API，你需要负责处理与数据库访问相关的所有事情，其中包含管理数据库资源和处理异常。如果你曾经使用JDBC往数据库中插入数据，那如下代码对你应该并不陌生：

程序清单**10.4**　使用**JDBC**在数据库里插入一行数据

```
private static final String SQL_INSERT_SPITTER =
 "insert into spitter (username, password, fullname) values (?, ?, ?)";
private DataSource dataSource;
public void addSpitter(Spitter spitter) {
  Connection conn = null;
  PreparedStatement stmt = null;
  try {
    conn = dataSource.getConnection();
    stmt = conn.prepareStatement(SQL_INSERT_SPITTER);
    stmt.setString(1, spitter.getUsername());
    stmt.setString(2, spitter.getPassword());
    stmt.setString(3, spitter.getFullName());
    stmt.execute();
  } catch (SQLException e) {
    // do something...not sure what, though
  } finally {
    try {
      if (stmt != null) {
        stmt.close();
      }
      if (conn != null) {
        conn.close();
      }
      } catch (SQLException e) {
        // I'm even less sure about what to do here
      }
    }
  }
```

获取连接
创建语句
绑定参数
执行语句
（以某种方式）处理异常
清理资源

看看这些失控的代码！这个超过20行的代码仅仅是为了向数据库中插入一个简单的对象。对于JDBC操作来讲，这应该是最简单的了。但

是为什么要用这么多行代码才能做如此简单的事情呢？实际上，并非如此，只有几行代码是真正用于进行插入数据的。但是JDBC要求你必须正确地管理连接和语句，并以某种方式处理可能抛出的SQLException异常。

再提一句这个SQLException异常：你不但不清楚如何处理它（因为并不知道哪里出错了），而且你还要捕捉它两次！你需要在插入记录出错时捕捉它，同时你还需要在关闭语句和连接出错的时候捕捉它。看起来我们要做很多的工作来处理可能出现的问题，而这些问题通常是难以通过编码来处理的。

再来看一下如下程序清单中的代码，我们使用传统的JDBC来更新数据库中Spitter表的一行。

程序清单**10.5**　使用**JDBC**更新数据库中的一行

```
private static final String SQL_UPDATE_SPITTER =
      "update spitter set username = ?, password = ?, fullname = ?"
      + "where id = ?";
public void saveSpitter(Spitter spitter) {
  Connection conn = null;
  PreparedStatement stmt = null;
  try {
    conn = dataSource.getConnection();                       <-- 获取连接
    stmt = conn.prepareStatement(SQL_UPDATE_SPITTER)         <-- 创建语句
    stmt.setString(1, spitter.getUsername());                <-- 绑定参数
    stmt.setString(2, spitter.getPassword());
    stmt.setString(3, spitter.getFullName());
    stmt.setLong(4, spitter.getId());
    stmt.execute();                                          <-- 执行语句
  } catch (SQLException e) {
    // Still not sure what I'm supposed to do here           <-- (以某种方式)处理异常
  } finally {
    try {
      if (stmt != null) {                                    <-- 清理资源
        stmt.close();
      }
      if (conn != null) {
        conn.close();
        }
      } catch (SQLException e) {
        // or here
      }
    }
  }
```

乍看上去，程序清单10.5和10.4是相同的。实际上，除了SQL字符串和创建语句的那一行，它们是完全相同的。同样，这里也使用大量代码来完成一件简单的事情，而且有很多重复的代码。在理想情况下，我们只需要编写与特定任务相关的代码。毕竟，这才是程序清单10.5

和10.4的不同之处，剩下的都是样板代码。

为了完成对JDBC的完整介绍，让我们看一下如何从数据库中获取数据。如下所示，它也不简单。

程序清单**10.6**　使用**JDBC**从数据库中查询一行数据

```
private static final String SQL_SELECT_SPITTER =
      "select id, username, fullname from spitter where id = ?";
public Spitter findOne(long id) {
  Connection conn = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;
  try {
    conn = dataSource.getConnection();                         获取连接
    stmt = conn.prepareStatement(SQL_SELECT_SPITTER);          创建语句
    stmt.setLong(1, id)                                        绑定参数
    rs = stmt.executeQuery();                                  执行语句
    Spitter spitter = null;
    if (rs.next()) {                                           处理结果
      spitter = new Spitter();
      spitter.setId(rs.getLong("id"));
      spitter.setUsername(rs.getString("username"));
      spitter.setPassword(rs.getString("password"));
      spitter.setFullName(rs.getString("fullname"));
    }
    return spitter;
  } catch (SQLException e) {                                   （以某种方式）处理异常
  } finally {                                                  清理资源
    if(rs != null) {
      try {
        rs.close();
      } catch(SQLException e) {}
    }

    if(stmt != null) {
      try {
      stmt.close();
      } catch(SQLException e) {}
    }

    if(conn != null) {
      try {
        conn.close();
      } catch(SQLException e) {}
    }
  }
  return null;
}
```

这段代码与插入和更新的样例一样冗长，甚至更为复杂。这就好像Pareto法则被倒了过来：只有20%的代码是真正用于查询数据的，而80%代码都是样板代码。

现在你可以看出，大量的JDBC代码都是用于创建连接和语句以及异常处理的样板代码。既然已经得出了这个观点，我们将不再接受它的折磨，以后你再也不会看到这样令人厌恶的代码了。

但实际上，这些样板代码是非常重要的。清理资源和处理错误确保了

数据访问的健壮性。如果没有它们的话，就不会发现错误而且资源也会处于打开的状态，这将会导致意外的代码和资源泄露。我们不仅需要这些代码，而且还要保证它是正确的。基于这样的原因，我们才需要框架来保证这些代码只写一次而且是正确的。

10.3.2 使用JDBC模板

Spring的JDBC框架承担了资源管理和异常处理的工作，从而简化了JDBC代码，让我们只需编写从数据库读写数据的必需代码。

正如前面小节所介绍过的，Spring将数据访问的样板代码抽象到模板类之中。Spring为JDBC提供了三个模板类供选择：

- `JdbcTemplate`：最基本的Spring JDBC模板，这个模板支持简单的JDBC数据库访问功能以及基于索引参数的查询；
- `NamedParameterJdbcTemplate`：使用该模板类执行查询时可以将值以命名参数的形式绑定到SQL中，而不是使用简单的索引参数；
- `SimpleJdbcTemplate`：该模板类利用Java 5的一些特性如自动装箱、泛型以及可变参数列表来简化JDBC模板的使用。

以前，在选择哪一个JDBC模板的时候，我们需要仔细权衡。但是从Spring 3.1开始，做这个决定变得容易多了。`SimpleJdbcTemplate`已经被废弃了，其Java 5的特性被转移到了`JdbcTemplate`中，并且只有在你需要使用命名参数的时候，才需要使用`NamedParameterJdbcTemplate`。这样的话，对于大多数的JDBC任务来说，`JdbcTemplate`就是最好的可选方案，这也是本小节中所关注的方案。

使用**JdbcTemplate**来插入数据

为了让`JdbcTemplate`正常工作，只需要为其设置`DataSource`就可以了，这使得在Spring中配置`JdbcTemplate`非常容易，如下面的`@Bean`方法所示：

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
}
```

在这里，`DataSource`是通过构造器参数注入进来的。这里所引用的`dataSource`bean可以是`javax.sql.DataSource`的任意实现，包

括我们在10.2小节中所创建的。

现在，我们可以将jdbcTemplate装配到Repository中并使用它来访问数据库。例如，SpitterRepository使用了JdbcTemplate：

```
@Repository
public class JdbcSpitterRepository implements SpitterRepository {

  private JdbcOperations jdbcOperations;

  @Inject
  public JdbcSpitterRepository(JdbcOperations jdbcOperations) {
    this.jdbcOperations = jdbcOperations;
  }

...

}
```

在这里，JdbcSpitterRepository类上使用了@Repository注解，这表明它将会在组件扫描的时候自动创建。它的构造器上使用了@Inject注解，因此在创建的时候，会自动获得一个JdbcOperations对象。JdbcOperations是一个接口，定义了JdbcTemplate所实现的操作。通过注入JdbcOperations，而不是具体的JdbcTemplate，能够保证JdbcSpitterRepository通过JdbcOperations接口达到与JdbcTemplate保持松耦合。

作为另外一种组件扫描和自动装配的方案，我们可以将JdbcSpitterRepository显式声明为Spring中的bean，如下所示：

```
@Bean
public SpitterRepository spitterRepository(JdbcTemplate jdbcTemplate) {
  return new JdbcSpitterRepository(jdbcTemplate);
}
```

在Repository中具备可用的JdbcTemplate后，我们可以极大地简化程序清单10.4中的addSpitter()方法。基于JdbcTemplate的addSpitter()方法如下：

程序清单**10.7**　基于**JdbcTemplate**的**addSpitter()**方法

```
public void addSpitter(Spitter spitter) {
    jdbcOperations.update(INSERT_SPITTER,          <—— 插入 Spitter

        spitter.getUsername(),
        spitter.getPassword(),
        spitter.getFullName(),
        spitter.getEmail(),
        spitter.isUpdateByEmail());
}
```

这个版本的addSpitter()方法简单多了。这里没有了创建连接和

语句的代码，也没有异常处理的代码，只剩下单纯的数据插入代码。

不能因为你看不到这些样板代码，就意味着它们不存在。样板代码被巧妙地隐藏到JDBC模板类中了。当update()方法被调用的时候JdbcTemplate将会获取连接、创建语句并执行插入SQL。

在这里，你也看不到对SQLException处理的代码。在内部，JdbcTemplate将会捕获所有可能抛出的SQLException，并将通用的SQLException转换为表10.1所列的那些更明确的数据访问异常，然后将其重新抛出。因为Spring的数据访问异常都是运行时异常，所以我们不必在addSpring ()方法中进行捕获。

使用JdbcTemplate来读取数据

JdbcTemplate也简化了数据的读取操作。程序清单10.8展现了新版本的findOne()方法，它使用了JdbcTemplate的回调，实现根据ID查询Spitter，并将结果集映射为Spitter对象。

程序清单**10.8**　使用**JdbcTemplate**查询**Spitter**

```
public Spitter findOne(long id) {
    return jdbcOperations.queryForObject(
            SELECT_SPITTER_BY_ID, new SpitterRowMapper(),
            id
            );
}
...
private static final class SpitterRowMapper
                 implements RowMapper<Spitter> {
  public Spitter mapRow(ResultSet rs, int rowNum)
                              throws SQLException {
    return new Spitter(
        rs.getLong("id"),
        rs.getString("username"),
        rs.getString("password"),
        rs.getString("fullName"),
        rs.getString("email"),
        rs.getBoolean("updateByEmail"));
  }
}
```

查询 Spitter

将查结果映射到对象

绑定参数

在这个findOne()方法中使用了JdbcTemplate的queryForObject()方法来从数据库查询Spitter。queryForObject()方法有三个参数：

- String对象，包含了要从数据库中查找数据的SQL；
- RowMapper对象，用来从ResultSet中提取数据并构建域对象（本例中为Spitter）；

- 可变参数列表，列出了要绑定到查询上的索引参数值。

真正奇妙的事情发生在SpitterRowMapper对象中，它实现了RowMapper接口。对于查询返回的每一行数据，JdbcTemplate将会调用RowMapper的mapRow()方法，并传入一个ResultSet和包含行号的整数。在SpitterRowMapper的mapRow()方法中，我们创建了Spitter对象并将ResultSet中的值填充进去。

就像addSpitter()那样，findOne()方法也不用写JDBC模板代码。不同于传统的JDBC，这里没有资源管理或者异常处理代码。使用JdbcTemplate的方法只需关注于如何从数据库中获取Spitter对象即可。

在**JdbcTemplate**中使用**Java 8**的**Lambda**表达式

因为RowMapper接口只声明了addRow()这一个方法，因此它完全符合函数式接口（functional interface）的标准。这意味着如果使用Java 8来开发应用的话，我们可以使用Lambda来表达RowMapper的实现，而不必再使用具体的实现类了。

例如，程序清单10.8中的findOne()方法可以使用Java 8的Lambda表达式改写，如下所示：

```
public Spitter findOne(long id) {
  return jdbcOperations.queryForObject(
    SELECT_SPITTER_BY_ID,
    (rs, rowNum) -&gt; {
      return new Spitter(
        rs.getLong("id"),
        rs.getString("username"),
        rs.getString("password"),
        rs.getString("fullName"),
        rs.getString("email"),
        rs.getBoolean("updateByEmail"));
    },
    id);
}
```

我们可以看到，Lambda表达式要比完整的RowMapper实现更为易读，不过它们的功能是相同的。Java会限制RowMapper中的Lambda表达式，使其满足所传入的参数。

另外，我们还可以使用Java 8的方法引用，在单独的方法中定义映射逻辑：

```
public Spitter findOne(long id) {
  return jdbcOperations.queryForObject(
    SELECT_SPITTER_BY_ID, this::mapSpitter, id);
}

private Spitter mapSpitter(ResultSet rs, int row) throws SQLException {
    return new Spitter(
      rs.getLong("id"),
      rs.getString("username"),
      rs.getString("password"),
      rs.getString("fullName"),
      rs.getString("email"),
      rs.getBoolean("updateByEmail"));
}
```

不管采用哪种方式，我们都不必显式实现RowMapper接口，但是与实现RowMapper类似，我们所提供的Lambda表达式和方法必须要接受相同的参数，并返回相同的类型。

使用命名参数

在清单10.7的代码中，addSpitter()方法使用了索引参数。这意味着我们需要留意查询中参数的顺序，在将值传递给update()方法的时候要保持正确的顺序。如果在修改SQL时更改了参数的顺序，那我们还需要修改参数值的顺序。

除了这种方法之外，我们还可以使用命名参数。命名参数可以赋予SQL中的每个参数一个明确的名字，在绑定值到查询语句的时候就通过该名字来引用参数。例如，假设SQL_INSERT_SPITTER查询语句是这样定义的：

```
private static final String SQL_INSERT_SPITTER =
      "insert into spitter (username, password, fullname) " +
      "values (:username, :password, :fullname)";
```

使用命名参数查询，绑定值的顺序就不重要了，我们可以按照名字来绑定值。如果查询语句发生了变化导致参数的顺序与之前不一致，我们不需要修改绑定的代码。

NamedParameterJdbcTemplate是一个特殊的JDBC模板类，它支持使用命名参数。在Spring中，NamedParameterJdbcTemplate的声明方式与常规的JdbcTemplate几乎完全相同：

```
@Bean
public NamedParameterJdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new NamedParameterJdbcTemplate(dataSource);
}
```

在这里，我们

将NamedParameterJdbcOperations（NamedParameterJdbcT所实现的接口）注入到Repository中，用它来替代JdbcOperations。现在的addSpitter()方法如下所示：

程序清单**10.9**　使用**Spring JDBC**模板的命名参数功能

```
private static final String INSERT_SPITTER =
        "insert into Spitter " +
        "    (username, password, fullname, email, updateByEmail) " +
        "values " +
        "    (:username, :password, :fullname, :email, :updateByEmail)";

public void addSpitter(Spitter spitter) {
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("username", spitter.getUsername());     <-- 绑定参数
    paramMap.put("password", spitter.getPassword());
    paramMap.put("fullname", spitter.getFullName());
    paramMap.put("email", spitter.getEmail());
    paramMap.put("updateByEmail", spitter.isUpdateByEmail());

    jdbcOperations.update(INSERT_SPITTER, paramMap);     <-- 执行数据插入
}
```

这个版本的addSpitter()比前一版本的代码要长一些。这是因为命名参数是通过java.util.Map来进行绑定的。不过，每行代码都关注于往数据库中插入Spitter对象。这个方法的核心功能并不会被资源管理或异常处理这样的代码所充斥。

10.4　小结

数据是应用程序的血液。有些数据中心论者甚至主张数据即应用。鉴于数据的重要地位，以健壮、简单和清晰的方式开发应用程序的数据访问部分就显得举足轻重了。

在Java中，JDBC是与关系型数据库交互的最基本方式。但是按照规范，JDBC有些太笨重了。Spring能够解除我们使用JDBC中的大多数痛苦，包括消除样板式代码、简化JDBC异常处理，你所需要做的仅仅是关注要执行的SQL语句。

在本章中，我们学习了Spring对数据持久化的支持，以及Spring为JDBC所提供的基于模板的抽象，它能够极大地简化JDBC的使用。

在下一章中，我们会继续Spring数据持久化这一话题，将会学习Spring为Java持久化API所提供的功能。

第11章　使用对象-关系映射持久化数据

本章内容：

- 使用Spring和Hibernate
- 借助上下文Session，编写不依赖于Spring的Repository
- 通过Spring使用JPA
- 借助Spring Data实现自动化的JPA Repository

小时候，骑自行车是一件很有趣的事情，对吧？在清晨，我们骑车上学。放学后，我们游逛到朋友家。当天色渐晚之时，在父母的呼喊声中，我们骑车回家。那些日子真的很有意思！

后来，随着慢慢长大，现在我们所需要的不仅仅是一辆自行车了。有时，我们需要走很远的路去上班或需要装载一些生活用品，还有可能接送孩子去上足球课。如果生活在得克萨斯州的话，我们还必须需要一台空调。我们的需求超出了自行车的功能范围。

在数据持久化的世界中，JDBC就像自行车。对于份内的工作，它能很好地完成并且在一些特定的场景下表现出色。但随着应用程序变得越来越复杂，对持久化的需求也变得更复杂。我们需要将对象的属性映射到数据库的列上，并且需要自动生成语句和查询，这样我们就能从无休止的问号字符串中解脱出来。此外，我们还需要一些更复杂的特性：

- 延迟加载（*Lazy loading*）：随着我们的对象关系变得越来越复杂，有时候我们并不希望立即获取完整的对象间关系。举一个典型的例子，假设我们在查询一组`PurchaseOrder`对象，而每个对象中都包含一个`LineItem`对象集合。如果我们只关心`PurchaseOrder`的属性，那查询出`LineItem`的数据就毫无意义。而且这可能是开销很大的操作。延迟加载允许我们只在需要的时候获取数据。
- 预先抓取（*Eager fetching*）：这与延迟加载是相对的。借助于预先抓取，我们可以使用一个查询获取完整的关联对象。如果我们需要`PurchaseOrder`及其关联的`LineItem`对象，预先抓取的功能可以在一个操作中将它们全部从数据库中取出来，节省了多

次查询的成本。
- 级联（*Cascading*）：有时，更改数据库中的表会同时修改其他表。回到我们订购单的例子中，当删除`Order`对象时，我们希望同时在数据库中删除关联的`LineItem`。

一些可用的框架提供了这样的服务，这些服务的通用名称是对象/关系映射（object-relational mapping，ORM）。在持久层使用ORM工具，可以节省数千行的代码和大量的开发时间。ORM工具能够把你的注意力从容易出错的SQL代码转向如何实现应用程序的真正需求。

Spring对多个持久化框架都提供了支持，包括Hibernate、iBATIS、Java数据对象（Java Data Objects，JDO）以及Java持久化API（Java Persistence API，JPA）。与Spring对JDBC的支持那样，Spring对ORM框架的支持提供了与这些框架的集成点以及一些附加的服务：

- 支持集成Spring声明式事务；
- 透明的异常处理；
- 线程安全的、轻量级的模板类；
- DAO支持类；
- 资源管理。

本章没有足够的篇幅介绍Spring支持的全部ORM框架。其实这并不会有什么问题，因为Spring对不同ORM解决方案的支持是很相似的。一旦掌握了Spring对某种ORM框架的支持后，你可以轻松地切换到另一个框架。

在本章中，我们将会看到Spring如何与最常用的两种ORM方案集成：Hibernate和JPA。同时还会通过Spring Data JPA了解一下Spring Data项目。借助这种方式，我们不仅可以学习到如何借助Spring Data JPA移除JPA Repository中的样板式代码，还能为下一章的如何将Spring Data用于无模式的存储打下基础。

让我们先来看看Spring是如何为Hibernate提供支持的。

11.1　在Spring中集成Hibernate

Hibernate是在开发者社区很流行的开源持久化框架。它不仅提供了基本的对象关系映射，还提供了ORM工具所应具有的所有复杂功能，比如缓存、延迟加载、预先抓取以及分布式缓存。

在本章中，我们会关注Spring如何与Hibernate集成，而不会涉及太多Hibernate使用时的复杂细节。如果你需要了解更多关于Hibernate如何使用的知识，我推荐你阅读Christian Bauer、Gavin King和Gary Gregory撰写的《*Java Persistence with Hibernate，Second Edition*》（Manning，2014，www.manning.com/bauer3/）或访问Hibernate的网站http://www.hibernate.org。

11.1.1 声明Hibernate的Session工厂

使用Hibernate所需的主要接口是`org.hibernate.Session`。`Session`接口提供了基本的数据访问功能，如保存、更新、删除以及从数据库加载对象的功能。通过Hibernate的`Session`接口，应用程序的Repository能够满足所有的持久化需求。

获取Hibernate Session对象的标准方式是借助于Hibernate `SessionFactory`接口的实现类。除了一些其他的任务，`SessionFactory`主要负责Hibernate `Session`的打开、关闭以及管理。

在Spring中，我们要通过Spring的某一个Hibernate Session工厂bean来获取Hibernate `SessionFactory`。从3.1版本开始，Spring提供了三个Session工厂bean供我们选择：

- `org.springframework.orm.hibernate3.LocalSession`
- `org.springframework.orm.hibernate3.annotation.A`
- `org.springframework.orm.hibernate4.LocalSession`

这些Session工厂bean都是Spring `FactoryBean`接口的实现，它们会产生一个Hibernate`SessionFactory`，它能够装配进任何`SessionFactory`类型的属性中。这样的话，就能在应用的Spring上下文中，与其他的bean一起配置Hibernate Session工厂。

至于选择使用哪一个Session工厂，这取决于使用哪个版本的Hibernate以及你使用XML还是使用注解来定义对象-数据库之间的映射关系。如果你使用Hibernate 3.2或更高版本（直到Hibernate 4.0，但不包含这个版本）并且使用XML定义映射的话，那么你需要定义Spring的`org.springframework.orm.hibernate3`包中的`LocalSessionFactoryBean`：

```
@Bean
public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
  LocalSessionFactoryBean sfb = new LocalSessionFactoryBean();
  sfb.setDataSource(dataSource);
  sfb.setMappingResources(new String[] { "Spitter.hbm.xml" });
  Properties props = new Properties();
  props.setProperty("dialect", "org.hibernate.dialect.H2Dialect");
  sfb.setHibernateProperties(props);
  return sfb;
}
```

在配置LocalSessionFactoryBean时，我们使用了三个属性。属性dataSource装配了一个DataSource bean的引用。属性mappingResources列出了一个或多个的Hibernate映射文件，在这些文件中定义了应用程序的持久化策略。最后，hibernateProperties属性配置了Hibernate如何进行操作的细节。在本示例中，我们配置Hibernate使用H2数据库并且要按照H2Dialect来构建SQL。

如果你更倾向于使用注解的方式来定义持久化，并且你还没有使用Hibernate 4的话，那么需要使用AnnotationSessionFactoryBean来代替LocalSessionFactoryBean：

```
@Bean
public AnnotationSessionFactoryBean sessionFactory(DataSource ds) {
  AnnotationSessionFactoryBean sfb = new AnnotationSessionFactoryBean();
  sfb.setDataSource(ds);
  sfb.setPackagesToScan(new String[] { "com.habuma.spittr.domain" });
  Properties props = new Properties();
  props.setProperty("dialect", "org.hibernate.dialect.H2Dialect");
  sfb.setHibernateProperties(props);
  return sfb;
}
```

如果你使用Hibernate 4的话，那么就应该使用org.springframework.orm.hibernate4中的LocalSessionFactoryBean。尽管它与Hibernate 3包中的LocalSessionFactoryBean使用了相同的名称，但是Spring 3.1新引入的这个Session工厂类似于Hibernate 3中LocalSessionFactoryBean和AnnotationSessionFactoryBean的结合体。它有很多相同的属性，能够支持基于XML的映射和基于注解的映射。如下的代码展现了如何对它进行配置，使其支持基于注解的映射：

```
@Bean
public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
  LocalSessionFactoryBean sfb = new LocalSessionFactoryBean();
  sfb.setDataSource(dataSource);
  sfb.setPackagesToScan(new String[] { "com.habuma.spittr.domain" });
  Properties props = new Properties();
  props.setProperty("dialect", "org.hibernate.dialect.H2Dialect");
  sfb.setHibernateProperties(props);
  return sfb;
}
```

在这两个配置中，dataSource和hibernateProperties属性都声明了从哪里获取数据库连接以及要使用哪一种数据库。这里不再列出Hibernate配置文件，而是使用packagesToScan属性告诉Spring扫描一个或多个包以查找域类，这些类通过注解的方式表明要使用Hibernate进行持久化，这些类可以使用的注解包括JPA的@Entity或@MappedSuperclass以及Hibernate的@Entity。

如果愿意的话，你还可以使用annotatedClasses属性来将应用程序中所有的持久化类以全限定名的方式明确列出：

```
sfb.setAnnotatedClasses(
  new Class<?>[] { Spitter.class, Spittle.class }
);
```

annotatedClasses属性对于准确指定少量的域类是不错的选择。如果你有很多的域类并且不想将其全部列出，又或者你想自由地添加或移除域类而不想修改Spring配置的话，那使用packagesToScan属性是更合适的。

在Spring应用上下文中配置完Hibernate的Session工厂bean后，那我们就可以创建自己的Repository类了。

11.1.2　构建不依赖于Spring的Hibernate代码

在Spring和Hibernate的早期岁月中，编写Repository类将会涉及到使用Spring的HibernateTemplate。HibernateTemplate能够保证每个事务使用同一个Session。但是这种方式的弊端在于我们的Repository实现会直接与Spring耦合。

现在的最佳实践是不再使用HibernateTemplate，而是使用上下文Session（Contextual session）。通过这种方式，会直接将Hibernate SessionFactory装配到Repository中，并使用它来获取Session，如下面的程序清单所示。

程序清单**11.1**　借助**Hibernate Session**实现不依赖于**Spring**的

Repository

```
public HibernateSpitterRepository(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;                    <—注入 SessionFactory
    }

    private Session currentSession() {
        return sessionFactory.getCurrentSession();           <—从 SessionFactory 中获取当前 Session
    }

    public long count() {
        return findAll().size();
    }

    public Spitter save(Spitter spitter) {
        Serializable id = currentSession().save(spitter);     <—使用当前 Session

        return new Spitter((Long) id,
                spitter.getUsername(),
                spitter.getPassword(),
                spitter.getFullName(),
                spitter.getEmail(),
                spitter.isUpdateByEmail());
    }
    public Spitter findOne(long id) {
        return (Spitter) currentSession().get(Spitter.class, id);
    }

    public Spitter findByUsername(String username) {
        return (Spitter) currentSession()
                .createCriteria(Spitter.class)
                .add(Restrictions.eq("username", username))
                .list().get(0);
    }

    public List<Spitter> findAll() {
        return (List<Spitter>) currentSession()
                .createCriteria(Spitter.class).list();
    }

}
```

在程序清单11.1中有几个地方需要注意。首先，我们通过`@Inject`注解让Spring自动将一个`SessionFactory`注入到`HibernateSpitterRepository`的`sessionFactory`属性中。接下来，在`currentSession()`方法中，我们使用这个`SessionFactory`来获取当前事务的Session。

另外需要注意的是，我们在类上使用了`@Repository`注解，这会为我们做两件事情。首先，`@Repository`是Spring的另一种构造性注解，它能够像其他注解一样被Spring的组件扫描所扫描到。这样就不必明确声明`HibernateSpitterRepository` bean了，只要这个Repository类在组件扫描所涵盖的包中即可。

除了帮助减少显式配置以外，`@Repository`还有另外一个用处。让我们回想一下模板类，它有一项任务就是捕获平台相关的异常，然后使用Spring统一非检查型异常的形式重新抛出。如果我们使用

Hibernate上下文Session而不是Hibernate模板的话，那异常转换会怎么处理呢？

为了给不使用模板的Hibernate Repository添加异常转换功能，我们只需在Spring应用上下文中添加一个`PersistenceExceptionTranslationPostProcessor` bean：

```
@Bean
public BeanPostProcessor persistenceTranslation() {
  return new PersistenceExceptionTranslationPostProcessor();
}
```

`PersistenceExceptionTranslationPostProcessor`是一个bean 后置处理器（bean post-processor），它会在所有拥有`@Repository`注解的类上添加一个通知器（advisor），这样就会捕获任何平台相关的异常并以Spring非检查型数据访问异常的形式重新抛出。

现在，Hibernate版本的Repository已经完成了。我们开发时，没有依赖Spring的特定类（除了`@Repository`注解以外）。这种不使用模板的方式也适用于开发纯粹的基于JPA的Repository，让我们再尝试开发另一个`SpitterRepository`实现类，这次我们使用的是JPA。

11.2 Spring与Java持久化API

Java持久化API（Java Persistence API，JPA）诞生在EJB 2实体Bean的废墟之上，并成为下一代Java持久化标准。JPA是基于POJO的持久化机制，它从Hibernate和Java数据对象（Java Data Object，JDO）上借鉴了很多理念并加入了Java 5注解的特性。

在Spring 2.0版本中，Spring首次集成了JPA的功能。具有讽刺意味的是，很多人批评（或赞赏）Spring颠覆了EJB。但是，当Spring支持JPA后，很多开发人员都推荐在基于Spring的应用程序中使用JPA实现持久化。实际上，有些人还将Spring-JPA的组合称为POJO开发的梦之队。

在Spring中使用JPA的第一步是要在Spring应用上下文中将实体管理器工厂（entity manager factory）按照bean的形式来进行配置。

11.2.1 配置实体管理器工厂

简单来讲，基于JPA的应用程序需要使用EntityManagerFactory的实现类来获取EntityManager实例。JPA定义了两种类型的实体管理器：

- 应用程序管理类型（*Application-managed*）：当应用程序向实体管理器工厂直接请求实体管理器时，工厂会创建一个实体管理器。在这种模式下，程序要负责打开或关闭实体管理器并在事务中对其进行控制。这种方式的实体管理器适合于不运行在Java EE容器中的独立应用程序。
- 容器管理类型（*Container-managed*）：实体管理器由Java EE创建和管理。应用程序根本不与实体管理器工厂打交道。相反，实体管理器直接通过注入或JNDI来获取。容器负责配置实体管理器工厂。这种类型的实体管理器最适用于Java EE容器，在这种情况下会希望在persistence.xml指定的JPA配置之外保持一些自己对JPA的控制。

以上的两种实体管理器实现了同一个EntityManager接口。关键的区别不在于EntityManager本身，而是在于EntityManager的创建和管理方式。应用程序管理类型的EntityManager是由EntityManagerFactory创建的，而后者是通过PersistenceProvider的createEntityManagerFactory()方法得到的。与此相对，容器管理类型的Entity ManagerFactory是通过PersistenceProvider的createContainerEntityManager Factory()方法获得的。

这对想使用JPA的Spring开发者来说又意味着什么呢？其实这并没太大的关系。不管你希望使用哪种EntityManagerFactory，Spring都会负责管理EntityManager。如果你使用的是应用程序管理类型的实体管理器，Spring承担了应用程序的角色并以透明的方式处理EntityManager。在容器管理的场景下，Spring会担当容器的角色。

这两种实体管理器工厂分别由对应的Spring工厂Bean创建：

- LocalEntityManagerFactoryBean生成应用程序管理类型的EntityManager-Factory；
- LocalContainerEntityManagerFactoryBean生成容器管理类型的Entity-ManagerFactory。

需要说明的是，选择应用程序管理类型的还是容器管理类型的EntityManager Factory，对于基于Spring的应用程序来讲是完

全透明的。当组合使用Spring和JPA时，处理EntityManagerFactory的复杂细节被隐藏了起来，数据访问代码只需关注它们的真正目标即可，也就是数据访问。

应用程序管理类型和容器管理类型的实体管理器工厂之间唯一值得关注的区别是在Spring应用上下文中如何进行配置。让我们先看看如何在Spring中配置应用程序管理类型的LocalEntityManagerFactoryBean，然后再看看如何配置容器管理类型的LocalContainerEntityManagerFactoryBean。

配置应用程序管理类型的**JPA**

对于应用程序管理类型的实体管理器工厂来说，它绝大部分配置信息来源于一个名为persistence.xml的配置文件。这个文件必须位于类路径下的META-INF目录下。

persistence.xml的作用在于定义一个或多个持久化单元。持久化单元是同一个数据源下的一个或多个持久化类。简单来讲，persistence.xml列出了一个或多个的持久化类以及一些其他的配置如数据源和基于XML的配置文件。如下是一个典型的persistence.xml文件，它是用于Spittr应用程序的：

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
      version="1.0">
    <persistence-unit name="spitterPU">
      <class>com.habuma.spittr.domain.Spitter</class>
      <class>com.habuma.spittr.domain.Spittle</class>
      <properties>
        <property name="toplink.jdbc.driver"
            value="org.hsqldb.jdbcDriver" />
        <property name="toplink.jdbc.url" value=
            "jdbc:hsqldb:hsql://localhost/spitter/spitter" />
        <property name="toplink.jdbc.user"
            value="sa" />
        <property name="toplink.jdbc.password"
            value="" />
      </properties>
    </persistence-unit>
  </persistence>
```

因为在persistence.xml文件中包含了大量的配置信息，所以在Spring中需要配置的就很少了。可以通过以下的@Bean注解方法在Spring中声明LocalEntityManagerFactoryBean：

```
@Bean
public LocalEntityManagerFactoryBean entityManagerFactoryBean() {
  LocalEntityManagerFactoryBean emfb
      = new LocalEntityManagerFactoryBean();
  emfb.setPersistenceUnitName("spitterPU");
  return emfb;
}
```

赋给persistenceUnitName属性的值就是persistence.xml中持久化单元的名称。

创建应用程序管理类型的EntityManagerFactory都是在persistence.xml中进行的，而这正是应用程序管理的本意。在应用程序管理的场景下（不考虑Spring时），完全由应用程序本身来负责获取EntityManagerFactory，这是通过JPA实现的PersistenceProvider做到的。如果每次请求EntityManagerFactory时都需要定义持久化单元，那代码将会迅速膨胀。通过将其配置在persistence.xml中，JPA就能够在这个特定的位置查找持久化单元定义了。

但借助于Spring对JPA的支持，我们不再需要直接处理PersistenceProvider了。因此，再将配置信息放在persistence.xml中就显得不那么明智了。实际上，这样做妨碍了我们在Spring中配置EntityManagerFactory（如果不是这样的话，我们可以提供一个Spring配置的数据源）。

鉴于以上的原因，让我们关注一下容器管理的JPA：

使用容器管理类型的JPA

容器管理的JPA采取了一个不同的方式。当运行在容器中时，可以使用容器（在我们的场景下是Spring）提供的信息来生成EntityManagerFactory。

你可以将数据源信息配置在Spring应用上下文中，而不是在persistence.xml中了。例如，如下的@Bean注解方法声明了在Spring中如何使用LocalContainerEntity-ManagerFactoryBean来配置容器管理类型的JPA：

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(
        DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {
  LocalContainerEntityManagerFactoryBean emfb =
      new LocalContainerEntityManagerFactoryBean();
  emfb.setDataSource(dataSource);
  emfb.setJpaVendorAdapter(jpaVendorAdapter);
  return emfb;
}
```

这里，我们使用了Spring配置的数据源来设置dataSource属性。任何javax.sql.DataSource的实现都是可以的。尽管数据源还可以在persistence.xml中进行配置，但是这个属性指定的数据源具有更高的优先级。

jpaVendorAdapter属性用于指明所使用的是哪一个厂商的JPA实现。Spring提供了多个JPA厂商适配器：

- EclipseLinkJpaVendorAdapter
- HibernateJpaVendorAdapter
- OpenJpaVendorAdapter
- TopLinkJpaVendorAdapter（在Spring 3.1版本中，已经将其废弃了）

在本例中，我们使用Hibernate作为JPA实现，所以将其配置为Hibernate-JpaVendorAdapter：

```
@Bean
public JpaVendorAdapter jpaVendorAdapter() {
  HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
  adapter.setDatabase("HSQL");
  adapter.setShowSql(true);
  adapter.setGenerateDdl(false);
  adapter.setDatabasePlatform("org.hibernate.dialect.HSQLDialect");
  return adapter;
}
```

有多个属性需要设置到厂商适配器上，但是最重要的是database属性，在上面我们设置了要使用的数据库是Hypersonic。这个属性支持的其他值如表11.1所示。

表**11.1　Hibernate**的**JPA**适配器支持多种数据库，可以通过其**database**属性配置使用哪个数据库

数据库平台	属性**database**的值
IBM DB2	DB2

Apache Derby	DERBY
H2	H2
Hypersonic	HSQL
Informix	INFORMIX
MySQL	MYSQL
Oracle	ORACLE
PostgresQL	POSTGRESQL
Microsoft SQL Server	SQLSERVER
Sybase	SYBASE

一些特定的动态持久化功能需要对持久化类按照指令（instrumentation）进行修改才能支持。在属性延迟加载（只在它们被实际访问时才从数据库中获取）的对象中，必须要包含知道如何查询未加载数据的代码。一些框架使用动态代理实现延迟加载，而有一些框架像JDO，则是在编译时执行类指令。

选择哪一种实体管理器工厂主要取决于如何使用它。但是，下面的小技巧可能会让你更加倾向于使用`LocalContainerEntityManagerFactoryBean`。

persistence.xml文件的主要作用就在于识别持久化单元中的实体类。但是从Spring 3.1开始，我们能够在`LocalContainerEntityManagerFactoryBean`中直接设置`packagesToScan`属性：

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(
        DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {
  LocalContainerEntityManagerFactoryBean emfb =
      new LocalContainerEntityManagerFactoryBean();
  emfb.setDataSource(dataSource);
  emfb.setJpaVendorAdapter(jpaVendorAdapter);
  emfb.setPackagesToScan("com.habuma.spittr.domain");
  return emfb;
}
```

在这个配置中，LocalContainerEntityManagerFactoryBean会扫描com.habuma.spittr.domain包，查找带有@Entity注解的类。因此，没有必要在persistence.xml文件中进行声明了。同时，因为DataSource也是注入到LocalContainer-EntityManagerFactoryBean中的，所以也没有必要在persistence.xml文件中配置数据库信息了。那么结论就是，persistence.xml文件完全没有必要存在了！你尽可以将其删除，让LocalContainerEntityManagerFactoryBean来处理这些事情。

从JNDI获取实体管理器工厂

还有一件需要注意的事项，如果将Spring应用程序部署在应用服务器中，EntityManagerFactory可能已经创建好了并且位于JNDI中等待查询使用。在这种情况下，可以使用Spring jee命名空间下的<jee:jndi-lookup>元素来获取对EntityManagerFactory的引用：

```
<jee:jndi-lookup id="emf" jndi-name="persistence/spitterPU" />
```

我们也可以使用如下的Java配置来获取EntityManagerFactory：

```
@Bean
public JndiObjectFactoryBean entityManagerFactory() {}
JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();
  jndiObjectFB.setJndiName("jdbc/SpittrDS");
  return jndiObjectFB;
}
```

尽管这种方法没有返回EntityManagerFactory，但是它的结果就是一个EntityManagerFactory bean。这是因为它所返回的JndiObjectFactoryBean是FactoryBean接口的实现，它能够创建EntityManagerFactory。

不管你采用何种方式得到EntityManagerFactory，一旦得到这样的对象，接下来就可以编写Repository了。让我们开始吧。

11.2.2 编写基于JPA的Repository

正如Spring对其他持久化方案的集成一样，Spring对JPA集成也提供了JpaTemplate模板以及对应的支持类JpaDaoSupport。但是，为了实现更纯粹的JPA方式，基于模板的JPA已经被弃用了。这与我们在11.1.2小节使用的Hibernate上下文Session是很类似的。

鉴于纯粹的JPA方式远胜于基于模板的JPA，所以在本节中我们将会重点关注如何构建不依赖Spring的JPA Repository。如下程序清单中的JpaSpitterRepository展现了如何开发不使用Spring JpaTemplate的JPA Repository。

程序清单**11.2** 不使用**Spring**模板的纯**JPA Repository**

```
package com.habuma.spittr.persistence;
  import java.util.List;
  import javax.persistence.EntityManagerFactory;
  import javax.persistence.PersistenceUnit;
  import org.springframework.dao.DataAccessException;
  import org.springframework.stereotype.Repository;
  import org.springframework.transaction.annotation.Transactional;
  import com.habuma.spittr.domain.Spitter;
  import com.habuma.spittr.domain.Spittle;

  @Repository
  @Transactional
  public class JpaSpitterRepository implements SpitterRepository {

    @PersistenceUnit                                   注入
    private EntityManagerFactory emf;              <-- EntityManagerFactory

    public void addSpitter(Spitter spitter) {
      emf.createEntityManager().persist(spitter);  <-- 创建并使用
    }                                                  EntityManager

    public Spitter getSpitterById(long id) {
      return emf.createEntityManager().find(Spitter.class, id);
    }

    public void saveSpitter(Spitter spitter) {
      emf.createEntityManager().merge(spitter);
    }
  ...
  }
```

程序清单11.2中，需要注意的是EntityManagerFactory属性，它使用了@PersistenceUnit注解，因此，Spring会将EntityManagerFactory注入到Repository之中。有了EntityManagerFactory之后，JpaSpitterRepository的方法就能使用它来创建EntityManager了，然后EntityManager可以针对数据库执行操作。

在JpaSpitterRepository中，唯一的问题在于每个方法都会调

用createEntityManager()。除了引入易出错的重复代码以外，这还意味着每次调用Repository的方法时，都会创建一个新的EntityManager。这种复杂性源于事务。如果我们能够预先准备好EntityManager，那会不会更加方便呢？

这里的问题在于EntityManager并不是线程安全的，一般来讲并不适合注入到像Repository这样共享的单例bean中。但是，这并不意味着我们没有办法要求注入EntityManager。如下的程序清单展现了如何借助@PersistentContext注解为JpaSpitterRepository设置EntityManager。

程序清单**11.3**　将**EntityManager**的代理注入到**Repository**之中

```
package com.habuma.spittr.persistence;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import com.habuma.spittr.domain.Spitter;
import com.habuma.spittr.domain.Spittle;

@Repository
@Transactional
public class JpaSpitterRepository implements SpitterRepository {

  @PersistenceContext
  private EntityManager em;                 <— 注入 EntityManager

  public void addSpitter(Spitter spitter) {
    em.persist(spitter);                    <— 使用 EntityManager
  }

  public Spitter getSpitterById(long id) {
    return em.find(Spitter.class, id);
  }

  public void saveSpitter(Spitter spitter) {
    em.merge(spitter);
  }
...
}
```

在这个新版本的JpaSpitterRepository中，直接为其设置了EntityManager，这样的话，在每个方法中就没有必要再通过EntityManagerFactory创建EntityManager了。尽管这种方式非常便利，但是你可能会担心注入的EntityManager会有线程安全性的问题。

这里的真相是@PersistenceContext并不会真正注入EntityManager——至少，精确来讲不是这样的。它没有将真正的EntityManager设置给Repository，而是给了它一个EntityManager的代理。真正的EntityManager是与当前事务

相关联的那一个，如果不存在这样的EntityManager的话，就会创建一个新的。这样的话，我们就能始终以线程安全的方式使用实体管理器。

另外，还需要了解@PersistenceUnit和@PersistenceContext并不是Spring的注解，它们是由JPA规范提供的。为了让Spring理解这些注解，并注入EntityManager Factory或EntityManager，我们必须要配置Spring的Persistence-AnnotationBeanPostProcessor。如果你已经使用了<context:annotation-config>或<context:component-scan>，那么你就不必再担心了，因为这些配置元素会自动注册PersistenceAnnotationBeanPostProcessor bean。否则的话，我们需要显式地注册这个bean：

```
@Bean
public PersistenceAnnotationBeanPostProcessor paPostProcessor() {
  return new PersistenceAnnotationBeanPostProcessor();
}
```

你可能也注意到了JpaSpitterRepository使用了@Repository和@Transactional注解。@Transactional表明这个Repository中的持久化方法是在事务上下文中执行的。

对于@Repository注解，它的作用与开发Hibernate上下文Session版本的Repository时是一致的。由于没有使用模板类来处理异常，所以我们需要为Repository添加@Repository注解，这样PersistenceExceptionTranslationPostProcessor就会知道要将这个bean产生的异常转换成Spring的统一数据访问异常。

既然提到了PersistenceExceptionTranslationPostProcessor，要记住的是我们需要将其作为一个bean装配到Spring中，就像我们在Hibernate样例中所做的那样：

```
@Bean
public BeanPostProcessor persistenceTranslation() {
  return new PersistenceExceptionTranslationPostProcessor();
}
```

提醒一下，不管对于JPA还是Hibernate，异常转换都不是强制要求的。如果你希望在Repository中抛出特定的JPA或Hibernate异常，只需将PersistenceException-TranslationPostProcessor省略掉即可，这样原来的异常就会正常地处理。但是，如果使用了Spring的异常转换，你会将所有的数据访问异常置于Spring的体系之下，这

样以后切换持久化机制的话会更容易。

11.3　借助Spring Data实现自动化的JPA Repository

尽管程序清单11.2和11.3程序清单中的方法都很简单，但它们依然还会直接与EntityManager交互来查询数据库。并且，仔细看一下的话，这些代码多少还是样板式的。例如，让我们重新审视addSpitter()方法：

```
public void addSpitter(Spitter spitter) {
    entityManager.persist(spitter);
  }
```

在任何具有一定规模的应用中，你可能会以几乎完全相同的方式多次编写这种方法。实际上，除了所持久化的Spitter对象不同以外，我敢打赌你以前肯定写过类似的方法。其实，JpaSpitterRepository中的其他方法也没有什么太大的创造性。领域对象会有所不同，但是所有Repository中的方法都是很通用的。

为什么我们需要一遍遍地编写相同的持久化方法呢，难道仅仅是因为要处理的领域类型不同吗？Spring Data JPA能够终结这种样板式的愚蠢行为。我们不再需要一遍遍地编写相同的Repository实现，Spring Data能够让我们只编写Repository接口就可以了。根本就不再需要实现类了。

例如，看一下SpitterRepository接口。

程序清单**11.4**　借助**Spring Data**，以接口定义的方式创建**Repository**

```
public interface SpitterRepository
        extends JpaRepository<Spitter, Long> {
}
```

此时，SpitterRepository看上去并没有什么作用。但是，它的功能远超出了表面上所看到的那样。

编写Spring Data JPA Repository的关键在于要从一组接口中挑选一个进行扩展。这里，SpitterRepository扩展了Spring Data JPA的JpaRepository（稍后，我会介绍几个其他的接口）。通过这种方

式，JpaRepository进行了参数化，所以它就能知道这是一个用来持久化Spitter对象的Repository，并且Spitter的ID类型为Long。另外，它还会继承18个执行持久化操作的通用方法，如保存Spitter、删除Spitter以及根据ID查询Spitter。

此时，你可能会想下一步就该编写一个类实现SpitterRepository和它的18个方法了。如果真的是这样的话，那本章就会变得乏味无聊了。其实，我们根本不需要编写SpitterRepository的任何实现类，相反，我们让Spring Data来为我们做这件事请。我们所需要做的就是对它提出要求。

为了要求Spring Data创建SpitterRepository的实现，我们需要在Spring配置中添加一个元素。如下的程序清单展现了在XML配置中启用Spring Data JPA所需要添加的内容：

程序清单**11.5**　配置**Spring Data JPA**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd">

    <jpa:repositories base-package="com.habuma.spittr.db" />

  ...

</beans>
```

<jpa:repositories>元素掌握了Spring Data JPA的所有魔力。就像<context:component-scan>元素一样，<jpa:repositories>元素也需要指定一个要进行扫描的base-package。不过，<context:component-scan>会扫描包（及其子包）来查找带有@Component注解的类，而<jpa:repositories>会扫描它的基础包来查找扩展自Spring Data JPA Repository接口的所有接口。如果发现了扩展自Repository的接口，它会自动生成（在应用启动的时候）这个接口的实现。

如果要使用Java配置的话，那就不需要使用<jpa:repositories>元素了，而是要在Java配置类上添加@EnableJpaRepositories注解。如下就是一个Java配置类，它使用了@EnableJpaRepositories注解，并且会扫描com.habuma.spittr.db包：

```
@Configuration
@EnableJpaRepositories(basePackages="com.habuma.spittr.db")
public class JpaConfiguration {
  ...
}
```

让我们回到SpitterRepository接口，它扩展自JpaRepository，而JpaRepository又扩展自Repository标记接口（虽然是间接的）。因此，SpitterRepository就传递性地扩展了Repository接口，也就是Repository扫描时所要查找的接口。当Spring Data找到它后，就会创建SpitterRepository的实现类，其中包含了继承自JpaRepository、PagingAndSortingRepository和CrudRepository的18个方法。

很重要的一点在于Repository的实现类是在应用启动的时候生成的，也就是Spring的应用上下文创建的时候。它并不是在构建时通过代码生成技术产生的，也不是接口方法调用时才创建的。

很漂亮的技术，对吧？

Spring Data JPA很棒的一点在于它能为Spitter对象提供18个便利的方法来进行通用的JPA操作，而无需你编写任何持久化代码。但是，如果你的需求超过了它所提供的这18个方法的话，该怎么办呢？幸好，Spring Data JPA提供了几种方式来为Repository添加自定义的方法。让我们看一下如何为Spring Data JPA编写自定义的查询方法。

11.3.1 定义查询方法

现在，SpitterRepository需要完成的一项功能是根据给定的username查找Spitter对象。比如，我们将SpitterRepository接口修改为如下所示的样子：

```
public interface SpitterRepository
       extends JpaRepository<Spitter, Long> {
    Spitter findByUsername(String username);
  }
```

这个新的findByUserName()非常简单，但是足以满足我们的需求。现在，该如何让Spring Data JPA提供这个方法的实现呢？

实际上，我们并不需要实现findByUsername()。方法签名已经告诉Spring Data JPA足够的信息来创建这个方法的实现了。

当创建Repository实现的时候，Spring Data会检查Repository接口的所有方法，解析方法的名称，并基于被持久化的对象来试图推测方法的目的。本质上，Spring Data定义了一组小型的领域特定语言（domain-specific language，DSL），在这里，持久化的细节都是通过Repository方法的签名来描述的。

Spring Data能够知道这个方法是要查找Spitter的，因为我们使用Spitter对JpaRepository进行了参数化。方法名findByUsername确定该方法需要根据username属性相匹配来查找Spitter，而username是作为参数传递到方法中来的。另外，因为在方法签名中定义了该方法要返回一个Spitter对象，而不是一个集合，因此它只会查找一个username属性匹配的Spitter。

findByUsername()方法非常简单，但是Spring Data也能处理更加有意思的方法名称。Repository方法是由一个动词、一个可选的主题（Subject）、关键词*By*以及一个断言所组成。在findByUsername()这个样例中，动词是*find*，断言是*Username*，主题并没有指定，暗含的主题是Spitter。

作为编写Repository方法名称的样例，我们参照名为readSpitterByFirstname-OrLastname()的方法，看一下方法中的各个部分是如何映射的。图11.1展现了这个方法是如何拆分的。

我们可以看到，这里的动词是*read*，与之前样例中的*find*有所差别。Spring Data允许在方法名中使用四种动词：*get*、*read*、*find*和*count*。其中，动词*get*、*read*和*find*是同义的，这三个动词对应的Repository方法都会查询数据并返回对象。而动词*count*则会返回匹配对象的数量，而不是对象本身。

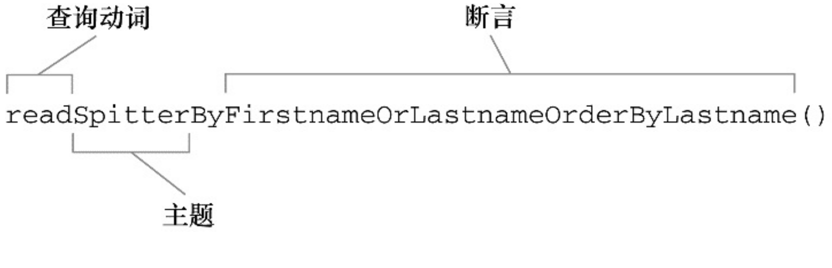

图11.1　Repository方法的命名遵循一种模式，有助于Spring Data生成针对数据库的查询

Repository方法的主题是可选的。它的主要目的是让你在命名方法的时候，有更多的灵活性。如果你更愿意将方法称

为readSpittersByFirstnameOrLastname()而不是readByFirstnameOrLastname()的话，那么你尽可以这么做。

对于大部分场景来说，主题会被省略掉。readSpittersByFirstnameOrLastname()与readPuppiesByFirstnameOrLastname()并没有什么差别，它们与readThose ThingsWeWantByFirstnameOrLastname()同样没有什么区别。要查询的对象类型是通过如何参数化JpaRepository接口来确定的，而不是方法名称中的主题。

在省略主题的时候，有一种例外情况。如果主题的名称以*Distinct*开头的话，那么在生成查询的时候会确保所返回结果集中不包含重复记录。

断言是方法名称中最为有意思的部分，它指定了限制结果集的属性。在readByFirstnameOrLastname()这个样例中，会通过firstname属性或lastname属性的值来限制结果。

在断言中，会有一个或多个限制结果的条件。每个条件必须引用一个属性，并且还可以指定一种比较操作。如果省略比较操作符的话，那么这暗指是一种相等比较操作。不过，我们也可以选择其他的比较操作，包括如下的种类：

- IsAfter、After、IsGreaterThan、GreaterThan
- IsGreaterThanEqual、GreaterThanEqual
- IsBefore、Before、IsLessThan、LessThan
- IsLessThanEqual、LessThanEqual
- IsBetween、Between
- IsNull、Null
- IsNotNull、NotNull
- IsIn、In
- IsNotIn、NotIn
- IsStartingWith、StartingWith、StartsWith
- IsEndingWith、EndingWith、EndsWith
- IsContaining、Containing、Contains
- IsLike、Like
- IsNotLike、NotLike
- IsTrue、True
- IsFalse、False
- Is、Equals
- IsNot、Not

要对比的属性值就是方法的参数。完整的方法签名如下所示：

```
List<Spitter> readByFirstnameOrLastname(String first, String last);
```

要处理String类型的属性时，条件中可能还会包含IgnoringCase或IgnoresCase，这样在执行对比的时候就会不再考虑字符是大写还是小写。例如，要在firstname和lastname属性上忽略大小写，那么可以将方法签名改成如下的形式：

```
List<Spitter> readByFirstnameIgnoringCaseOrLastnameIgnoresCase(
                    String first, String last);
```

需要注意，IgnoringCase和IgnoresCase是同义的，你可以随意挑选一个最合适的。

作为IgnoringCase/IgnoresCase的替代方案，我们还可以在所有条件的后面添加AllIgnoringCase或AllIgnoresCase，这样它就会忽略所有条件的大小写：

```
List<Spitter> readByFirstnameOrLastnameAllIgnoresCase(
                    String first, String last);
```

注意，参数的名称是无关紧要的，但是它们的顺序必须要与方法名称中的操作符相匹配。

最后，我们还可以在方法名称的结尾处添加OrderBy，实现结果集排序。例如，我们可以按照lastname属性升序排列结果集：

```
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAsc(
                    String first, String last);
```

如果要根据多个属性排序的话，只需将其依序添加到OrderBy中即可。例如，下面的样例中，首先会根据lastname升序排列，然后根据firstname属性降序排列：

```
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAscFirstnameDesc(
                    String first, String last);
```

可以看到，条件部分是通过And或者Or进行分割的。

我们不可能（至少很难）提供一个权威的列表，将使用Spring Data方法命名约定可以编写出来的方法种类全部列出来。但是，如下给出了几个符合方法命名约定的方法签名：

- `List<Pet> findPetsByBreedIn(List<String> breed)`
- `int countProductsByDiscontinuedTrue()`
- `List<Order> findByShippingDateBetween(Date start, Date end)`

我们只是初步体验了所能声明的方法种类，Spring Data JPA会为我们实现这些方法。现在，我们只需知道通过使用属性名和关键字构建Repository方法签名，就能让Spring Data JPA生成方法实现，完成几乎所有能够想象到的查询。

不过，Spring Data这个小型的DSL依旧有其局限性，有时候通过方法名称表达预期的查询很烦琐，甚至无法实现。如果遇到这种情形的话，Spring Data能够让我们通过`@Query`注解来解决问题。

11.3.2 声明自定义查询

假设我们想要创建一个Repository方法，用来查找E-mail地址是Gmail邮箱的`Spitter`。有一种方式就是定义一个`findByEmailLike()`方法，然后每次想查找Gmail用户的时候就将“%gmail.com”传递进来。不过，更好的方案是定义一个更加便利的`findAllGmailSpitters()`方法，这样的话，就不用将Email地址的一部分传递进来了：

```
List<Spitter> findAllGmailSpitters();
```

不过，这个方法并不符合Spring Data的方法命名约定。当Spring Data试图生成这个方法的实现时，无法将方法名的内容与Spitter元模型进行匹配，因此会抛出异常。

如果所需的数据无法通过方法名称进行恰当地描述，那么我们可以使用`@Query`注解，为Spring Data提供要执行的查询。对于`findAllGmailSpitters()`方法，我们可以按照如下的方式来使用`@Query`注解：

```
@Query("select s from Spitter s where s.email like '%gmail.com'")
List<Spitter> findAllGmailSpitters();
```

我们依然不需要编写`findAllGmailSpitters()`方法的实现，只需提供查询即可，让Spring Data JPA知道如何实现这个方法。

可以看到，当使用方法命名约定很难表达预期的查询时，`@Query`注

解能够发挥作用。如果按照命名约定，方法的名称特别长的时候，也可以使用这个注解。例如，考虑如下的查询方法：

```
List<Order>
  findByCustomerAddressZipCodeOrCustomerNameAndCustomerAddressState();
```

这真的是一个方法的名称！我不得不在返回类型后将其断开，这样才能适应本书页面的宽度。

我承认这是一个有点牵强的例子。但在现实世界中，确实存在这样的需求，使用Repository方法所执行的查询会得到一个很长的方法名。在这种情况下，你最好使用一个较短的方法名，并使用`@Query`来指定该方法要如何查询数据库。

对于Spring Data JPA的接口来说，`@Query`是一种添加自定义查询的便利方式。但是，它仅限于单个JPA查询。如果我们需要更为复杂的功能，无法在一个简单的查询中处理的话，该怎么办呢？

11.3.3 混合自定义的功能

有些时候，我们需要Repository所提供的功能是无法用Spring Data的方法命名约定来描述的，甚至无法用`@Query`注解设置查询来实现。尽管Spring Data JPA非常棒，但是它依然有其局限性，可能需要我们按照传统的方式来编写Repository方法：也就是直接使用`EntityManager`。当遇到这种情况的时候，我们是不是要放弃Spring Data JPA，重新按照11.2.2小节中的方式来编写Repository呢？

简单来说，是这样的。如果你需要做的事情无法通过Spring Data JPA来实现，那就必须要在一个比Spring Data JPA更低的层级上使用JPA。好消息是我们没有必要完全放弃Spring Data JPA。我们只需在必须使用较低层级JPA的方法上，才使用这种传统的方式即可，而对于Spring Data JPA知道该如何处理的功能，我们依然可以通过它来实现。

当Spring Data JPA为Repository接口生成实现的时候，它还会查找名字与接口相同，并且添加了`Impl`后缀的一个类。如果这个类存在的话，Spring Data JPA将会把它的方法与Spring Data JPA所生成的方法合并在一起。对于SpitterRepository接口而言，要查找的类名为`SpitterRepositoryImpl`。

为了阐述该功能，假设我们需要在`SpitterRepository`中添加一

个方法，发表Spittle数量在10,000及以上的Spitter将会更新为Elite状态。使用Spring Data JPA的方法命名约定或使用@Query均没有办法声明这样的方法。最为可行的方案是使用如下的eliteSweep()方法。

程序清单11.6　将活跃的Spitter用户升级为Elite状态的Repository方法

```
public class SpitterRepositoryImpl implements SpitterSweeper {

  @PersistenceContext
  private EntityManager em;

  public int eliteSweep() {
    String update =
        "UPDATE Spitter spitter " +
        "SET spitter.status = 'Elite' " +
        "WHERE spitter.status = 'Newbie' " +
        "AND spitter.id IN (" +
        "SELECT s FROM Spitter s WHERE (" +
        "  SELECT COUNT(spittles) FROM s.spittles spittles) > 10000" +
        ")";
    return em.createQuery(update).executeUpdate();
  }

}
```

我们可以看到，eliteSweep()方法与之前在11.2.2小节中所创建的Repository方法并没有太大的差别。SpitterRepositoryImpl没有什么特殊之处，它使用被注入的EntityManager来完成预期的任务。

注意，SpitterRepositoryImpl并没有实现SpitterRepository接口。Spring Data JPA负责实现这个接口。SpitterRepositoryImpl（将它与Spring Data的Repository关联起来的是它的名字）实现了SpitterSweeper接口，它如下所示：

```
public interface SpitterSweeper{
    int eliteSweep();
}
```

我们还需要确保eliteSweep()方法会被声明在SpitterRepository接口中。要实现这一点，避免代码重复的简单方式就是修改SpitterRepository，让它扩展SpitterSweeper：

```
public interface SpitterRepository
        extends JpaRepository<Spitter, Long>,
                SpitterSweeper {
    ....
}
```

如前所述，Spring Data JPA将实现类与接口关联起来是基于接口的名称。但是，Impl后缀只是默认的做法，如果你想使用其他后缀的话，只需在配置@EnableJpa-Repositories的时候，设置repositoryImplementationPostfix属性即可：

```
@EnableJpaRepositories(
  basePackages="com.habuma.spittr.db",
  repositoryImplementationPostfix="Helper")
```

如果在XML中使用<jpa:repositories>元素来配置Spring Data JPA的话，我们可以借助repository-impl-postfix属性指定后缀：

```
<jpa:repositories base-package="com.habuma.spittr.db"
    repository-impl-postfix="Helper" />
```

我们将后缀设置成了Helper，Spring Data JPA将会查找名为SpitterRepository-Helper的类，用它来匹配SpitterRepository接口。

11.4　小结

对于很多应用来讲，关系型数据库是主流的数据存储形式，并且这种情况已经持续了很多年。使用JDBC并且将对象映射为数据库表是很烦琐乏味的事情，像Hibernate和JPA这样的ORM方案能够让我们以更加声明式的模型实现数据持久化。尽管Spring没有为ORM提供直接的支持，但是它能够与多种流行的ORM方案集成，包括Hibernate与Java持久化API。

在本章中，我们看到了如何在Spring应用中使用Hibernate的上下文Session，这样我们的Repository就能包含很少甚至不包含Spring相关的代码。与之类似，我们还看到了如何将EntityManagerFactory或EntityManager注入到Repository实现中，从而实现不依赖于Spring的JPA Repository。

我们稍后初步了解了Spring Data，在这个过程中，只需声明JPA Repository接口即可，让Spring Data JPA在运行时自动生成这些接口的

实现。当我们需要的Repository方法超出了Spring Data JPA所提供的功能时，可以借助`@Query`注解以及编写自定义的Repository方法来实现。

但是，对于Spring Data的整体功能来说，我们只是接触到了皮毛。在下一章中，我们将会更加深入地学习Spring Data的方法命名DSL，以及Spring Data如何为关系型数据库以外的领域带来帮助。也就是说：我们将会看到Spring Data如何支持新兴的NoSQL数据库，这些数据库在最近几年变得越来越流行。

第12章　使用NoSQL数据库

本章内容：

- 为MongoDB和Neo4j编写Repository
- 为多种数据存储形式持久化数据
- 组合使用Spring和Redis

亨利·福特在他的自传中曾经写过一句很著名的话："任何顾客可以将这辆车漆成任何他所愿意的颜色，只要保持它的黑色就可以"[1]。有人说这句话是傲慢和固执的，而有些人则说这句话反映出了他的幽默。事实上，在这本自传出版的时候，他通过使用一种快速烘干的油漆降低了成本，而当时这种油漆只有黑色的。

福特的这句著名的话也可以用在数据库领域，多年来，我们一直被告知，我们可以使用任意想要的数据库，只要它是关系型数据库就行。关系型数据库已经垄断应用开发领域好多年了。

随着一些竞争者进入数据库领域，关系型数据库的垄断地位开始被弱化。所谓的"NoSQL"数据库开始侵入生产型的应用之中，我们也认识到并没有一种全能型的数据库。现在有了更多的可选方案，所以能够为要解决的问题选择最佳的数据库。

在前面的几章中，我们关注于关系型数据库，首先使用Spring对JDBC支持，然后使用对象-关系映射。在上一章，我们看到了Spring Data JPA，它是Spring Data项目下的多个子项目之一。通过在运行时自动生成Repository实现，Spring Data JPA能够让使用JPA的过程更加简单容易。

Spring Data还提供了对多种NoSQL数据库的支持，包括MongoDB、Neo4j和Redis。它不仅支持自动化的Repository，还支持基于模板的数据访问和映射注解。在本章中，将会看到如何为非关系型的NoSQL数据库编写Repository。首先，我们将从Spring Data MongoDB开始，看一下如何编写Repository来处理基于文档的数据。

12.1　使用MongoDB持久化文档数据

有一些数据的最佳表现形式是文档（document）。也就是说，不要把这些数据分散到多个表、节点或实体中，将这些信息收集到一个非规范化（也就是文档）的结构中会更有意义。尽管两个或两个以上的文档有可能会彼此产生关联，但是通常来讲，文档是独立的实体。能够按照这种方式优化并处理文档的数据库，我们称之为文档数据库。

例如，假设我们要编写一个应用程序来获取大学生的成绩单，可能需要根据学生的名字来查询其成绩单，或者根据一些通用的属性来查询成绩单。但是，每个学生是相互独立的，任意的两个成绩单之间没有必要相互关联。尽管我们能够使用关系型数据库模式来获取成绩单数据（也许你曾经这样做过），但文档型数据库可能才是更好的方案。

文档数据库不适用于什么场景

了解文档型数据库能够用于什么场景是很重要的。但是，知道文档型数据库在什么情况下不适用同样也是很重要的。文档数据库不是通用的数据库，它们所擅长解决的是一个很小的问题集。

有些数据具有明显的关联关系，文档型数据库并没有针对存储这样的数据进行优化。例如，社交网络表现了应用中不同的用户之间是如何建立关联的，这种情况就不适合放到文档型数据库中。在文档数据库中存储具有丰富关联关系的数据也并非完全不可能，但这样做的话，你通常会发现遇到的挑战要多于所带来的收益。

Spittr应用的域对象并不适合文档数据库。在本章中，我们将会在一个购物订单系统中学习MongoDB。

MongoDB是最为流行的开源文档数据库之一。Spring Data MongoDB提供了三种方式在Spring应用中使用MongoDB：

- 通过注解实现对象-文档映射；
- 使用`MongoTemplate`实现基于模板的数据库访问；
- 自动化的运行时Repository生成功能。

我们已经看到Spring Data JPA如何为基于JPA的数据访问实现自动化Repository生成功能。Spring Data MongoDB为基于MongoDB的数据访问提供了相同的功能。

不过，与Spring Data JPA不同的是，Spring Data MongoDB提供了将Java对象映射为文档的功能。（Spring Data JPA没有必要为JPA提供这样的注解，因为JPA规范本身就提供了对象-关系映射注解）。除此之

外，Spring Data MongoDB为通用的文档操作任务提供了基于模板的数据访问方式。

但是，在使用这些特性之前，我们首先要配置Spring Data MongoDB。

12.1.1　启用MongoDB

为了有效地使用Spring Data MongoDB，我们需要在Spring配置中添加几个必要的bean。首先，我们需要配置MongoClient，以便于访问MongoDB数据库。同时，我们还需要有一个MongoTemplate bean，实现基于模板的数据库访问。此外，不是必须，但是强烈推荐启用Spring Data MongoDB的自动化Repository生成功能。

如下的程序清单展现了如何编写简单的Spring Data MongoDB配置类，它包含了上述的几个bean：

程序清单**12.1　Spring Data MongoDB**的必要配置

```
package orders.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.core.MongoFactoryBean;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.repository.config.
                                        EnableMongoRepositories;
import com.mongodb.Mongo;

@Configuration
@EnableMongoRepositories(basePackages="orders.db")      <-- 启用 MongoDB 的 Repository 功能
public class MongoConfig {

    @Bean
    public MongoFactoryBean mongo() {                    <-- MongoClient bean
        MongoFactoryBean mongo = new MongoFactoryBean();
        mongo.setHost("localhost");
        return mongo;
    }

    @Bean
    public MongoOperations mongoTemplate(Mongo mongo) {  <-- MongoTemplate bean
        return new MongoTemplate(mongo, "OrdersDB");
    }

}
```

在上一章中，我们通过@EnableJpaRepositories注解，启用了Spring Data的自动化JPA Repository生成功能。与之类似，@EnableMongoRepositories为MongoDB实现了相同的功能。

除了@EnableMongoRepositories之外，程序清单12.1中还包含了两个带有@Bean注解的方法。第一个@Bean方法使

用MongoFactoryBean声明了一个Mongo实例。这个bean将Spring Data MongoDB与数据库本身连接了起来（与使用关系型数据时DataSource所做的事情并没有什么区别）。尽管我们可以使用MongoClient直接创建Mongo实例，但如果这样做的话，就必须要处理MongoClient构造器所抛出的UnknownHostException异常。在这里，使用Spring Data MongoDB的MongoFactoryBean更加简单。因为它是一个工厂bean，因此MongoFactoryBean会负责构建Mongo实例，我们不必再担心UnknownHostException异常。

另外一个@Bean方法声明了MongoTemplate bean，在它构造时，使用了其他@Bean方法所创建的Mongo实例的引用以及数据库的名称。稍后，你将会看到如何使用MongoTemplate来查询数据库。即便不直接使用MongoTemplate，我们也会需要这个bean，因为Repository的自动化生成功能在底层使用了它。

除了直接声明这些bean，我们还可以让配置类扩展AbstractMongo-Configuration并重载getDatabaseName()和mongo()方法。如下的程序清单展现了如何使用这种配置方式。

程序清单**12.2**　借助**@EnableMongoRepositories**启用**Spring Data MongoDB**

```
package orders.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.config.
                                   AbstractMongoConfiguration;
import org.springframework.data.mongodb.repository.config.
                                     EnableMongoRepositories;
import com.mongodb.Mongo;
import com.mongodb.MongoClient;

@Configuration
@EnableMongoRepositories("orders.db")
public class MongoConfig extends AbstractMongoConfiguration {

  @Override
  protected String getDatabaseName() {        <—— 指定数据库名称
    return "OrdersDB";
  }

  @Override
  public Mongo mongo() throws Exception {        <——创建 Mongo 客户端
    return new MongoClient();
  }

}
```

这个新的配置类与程序清单12.1的功能是相同的，只不过在篇幅上更加简洁。最为显著的区别在于这个配置中没有直接声明MongoTemplate bean，当然它还是会被隐式地创建。我们在这

里重载了getDatabaseName()方法来提供数据库的名称。mongo()方法依然会创建一个MongoClient的实例，因为它会抛出Exception，所以我们可以直接使用MongoClient，而不必再使用MongoFactoryBean了。

到目前为止，不管是使用程序清单12.1还是12.2，都为Spring Data MongoDB提供了一个运行配置，也就是说，只要MongoDB服务器运行在本地即可。如果MongoDB服务器运行在其他的机器上，那么可以在创建MongoClient的时候进行指定：

```
public Mongo mongo() throws Exception {
  return new MongoClient("mongodbserver");
}
```

另外，MongoDB服务器有可能监听的端口并不是默认的27017。如果是这样的话，在创建MongoClient的时候，还需要指定端口：

```
public Mongo mongo() throws Exception {
  return new MongoClient("mongodbserver", 37017);
}
```

如果MongoDB服务器运行在生产配置上，我认为你可能还启用了认证功能。在这种情况下，为了访问数据库，我们还需要提供应用的凭证。访问需要认证的MongoDB服务器稍微有些复杂，如下面的程序清单所示。

程序清单**12.3**　创建**MongoClient**来访问需要认证的**MongoDB**服务器

```
@Autowired
private Environment env;

@Override
public Mongo mongo() throws Exception {
  MongoCredential credential =
    MongoCredential.createMongoCRCredential(      <— 创建 MongoDB 凭证
        env.getProperty("mongo.username"),
        "OrdersDB",
        env.getProperty("mongo.password").toCharArray());

  return new MongoClient(                        <— 创建 MongoClient
       new ServerAddress("localhost", 37017),
      Arrays.asList(credential));
}
```

为了访问需要认证的MongoDB服务器，MongoClient在实例化的时候必须要有一个MongoCredential的列表。在程序清单12.3中，我们为此创建了一个MongoCredential。为了将凭证信息的细节放在配置类外边，它们是通过注入的Environment对象解析得到的。

为了使这个讨论更加完整，Spring Data MongoDB还支持通过XML来进行配置。你可能也知道，我更喜欢Java配置的方案。但是，如果你喜欢XML配置的话，如下的程序清单展现了如何使用`mongo`配置命名空间来配置Spring Data MongoDB。

程序清单**12.4　Spring Data MongoDB**提供了**XML**配置的方案

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="
    http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <mongo:repositories base-package="orders.db" />

    <mongo:mongo />

    <bean id="mongoTemplate"
        class="org.springframework.data.mongodb.core.MongoTemplate">
      <constructor-arg ref="mongo" />
      <constructor-arg value="OrdersDB" />
    </bean>

</beans>
```

声明mongo命名空间

声明Mongo Client

启用Repository生成功能

创建MongoTemplate bean

现在Spring Data MongoDB已经配置完成了，我们很快就可以使用它来保存和查询文档了。但首先，需要使用Spring Data MongoDB的对象-文档映射注解为Java领域对象建立到持久化文档的映射关系。

12.1.2　为模型添加注解，实现MongoDB持久化

当使用JPA的时候，我们需要将Java实体类映射到关系型表和列上。JPA规范提供了一些支持对象-关系映射的注解，而有一些JPA实现，如Hibernate，也添加了自己的映射注解。

但是，MongoDB并没有提供对象-文档映射的注解。Spring Data MongoDB填补了这一空白，提供了一些将Java类型映射为MongoDB文档的注解。表12.1描述了这些注解。

表**12.1**　用于对象-文档映射的**Spring Data MongoDB**注解

注　　解	描　　述
`@Document`	标示映射到MongoDB文档上的领域对象

`@Id`	标示某个域为ID域
`@DbRef`	标示某个域要引用其他的文档，这个文档有可能位于另外一个数据库中
`@Field`	为文档域指定自定义的元数据
`@Version`	标示某个属性用作版本域

`@Document`和`@Id`注解类似于JPA的`@Entity`和`@Id`注解。我们将会经常使用这两个注解，对于要以文档形式保存到MongoDB数据库的每个Java类型都会使用这两个注解。例如，如下的程序清单展现了如何为Order类添加注解，它会被持久化到MongoDB中。

程序清单**12.5　Spring Data MongoDB**注解将**Java**类型映射为文档

```
package orders;
import java.util.Collection;
import java.util.LinkedHashSet;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

@Document                                <-- 这是一个文档
 public class Order {

  @Id
  private String id;                       <-- 指定ID

  @Field("client")
  private String customer;                   <-- 覆盖默认的域名

  private String type;

  private Collection<Item> items = new LinkedHashSet<Item>();

  public String getCustomer() {
    return customer;
  }

  public void setCustomer(String customer) {
    this.customer = customer;
  }

  public String getType() {
    return type;
  }

  public void setType(String type) {
    this.type = type;
  }
  public Collection<Item> getItems() {
    return items;
  }

  public void setItems(Collection<Item> items) {
    this.items = items;
  }

  public String getId() {
    return id;
  }

}
```

我们可以看到，`Order`类添加了`@Document`注解，这样它就能够借助`MongoTemplate`或自动生成的Repository进行持久化。其`id`属性上使用了`@Id`注解，用来指定它作为文档的ID。除此之外，`customer`属性上使用了`@Field`注解，这样的话，当文档持久化的时候`customer`属性将会映射为名为`client`的域。

注意，其他的属性并没有添加注解。除非将属性设置为瞬时态（transient）的，否则Java对象中所有的域都会持久化为文档中的域。并且如果我们不使用`@Field`注解进行设置的话，那么文档域中的名字将会与对应的Java属性相同。

同时，需要注意的是`items`属性，它指的是订单中具体条目的集合。在传统的关系型数据库中，这些条目将会保存在另外的一个数据库表中，通过外键进行应用，`items`域上很可能还会使用JPA的

@OneToMany注解。但在这里，情形完全不同。

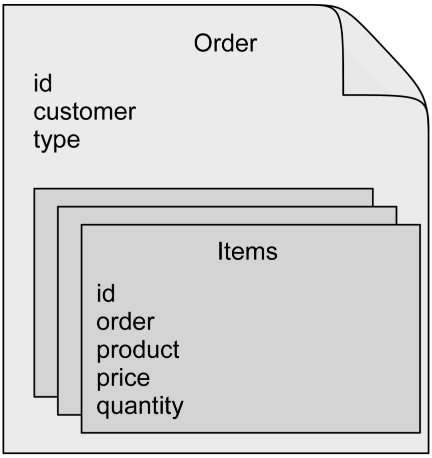

图12.1　文档展现了关联但非规范化的数据。
相关的概念（如订单中的条目）被嵌入到顶层的文档数据中

如我前面所述，文档可以与其他的文档产生关联，但这并不是文档数据库所擅长的功能。在本例购买订单与行条目之间的关联关系中，行条目只是同一个订单文档里面内嵌的一部分（如图12.1所示）。因此，没有必要为这种关联关系添加任何注解。实际上，Item类本身并没有任何注解：

```
package orders;

public class Item {

  private Long id;
  private Order order;
  private String product;
  private double price;
  private int quantity;
  public Order getOrder() {
    return order;
  }

  public String getProduct() {
    return product;
  }

  public void setProduct(String product) {
    this.product = product;
  }

  public double getPrice() {
    return price;
  }

  public void setPrice(double price) {
         this.price = price;
       }

       public int getQuantity() {
         return quantity;
       }

       public void setQuantity(int quantity) {
         this.quantity = quantity;
       }

       public Long getId() {
         return id;
       }

     }
```

我们没有必要为Item添加@Document注解，也没有必要为它的域指定@Id。这是因为我们不会单独将Item持久化为文档。它始终会是Order文档中Item列表的一个成员，并且会作为文档中的嵌入元素。

当然，如果你想指定Item中的某个域如何持久化到文档中，那么可以为对应的Item属性添加@Field注解。不过在本例中，并没有必要这样做。

我们现在已经为Java对象添加了MongoDB持久化的注解。接下来，看一下如何使用MongoTemplate来存储它们。

12.1.3 使用MongoTemplate访问MongoDB

我们已经在配置类中配置了MongoTemplate bean，不管是显式声明还是扩展AbstractMongoConfiguration都能实现相同的效果。接下来，需要做的就是将其注入到使用它的地方：

```
@Autowired
MongoOperations mongo;
```

注意，在这里我们将MongoTemplate注入到一个类型为MongoOperations的属性中。MongoOperations是MongoTemplate所实现的接口，不使用具体实现是一个好的做法，尤其是在注入的时候。

MongoOperations暴露了多个使用MongoDB文档数据库的方法。在这里，我们不可能讨论所有的方法，但是可以看一下最为常用的几个操作，比如计算文档集合中有多少条文档。使用注入的MongoOperations，我们可以得到Order集合并调用count()来得到数量：

```
long orderCount = mongo.getCollection("order").count();
```

现在，假设要保存一个新的Order。为了完成这个任务，我们可以调用save()方法：

```
Order order = new Order();
... // set properties and add line items
mongo.save(order, "order");
```

save()方法的第一个参数是新创建的Order，第二个参数是要保存的文档存储的名称。

另外，我们还可以调用findById()方法来根据ID查找订单：

```
String orderId = ...;
Order order = mongo.findById(orderId, Order.class);
```

对于更高级的查询，我们需要构造Query对象并将其传递给find()方法。例如，要查找所有client域等于“Chuck Wagon”的订单，可以使用如下的代码：

```
List<Order> chucksOrders = mongo.find(Query.query(
    Criteria.where("client").is("Chuck Wagon")), Order.class);
```

在本例中，用来构造Query对象的Criteria只检查了一个域，但是它也可以用来构造更加有意思的查询。比如，我们想要查询Chuck所有通过Web创建的订单：

```
List<Order> chucksWebOrders = mongo.find(Query.query(
    Criteria.where("customer").is("Chuck Wagon")
            .and("type").is("WEB")), Order.class);
```

如果你想移除某一个文档的话，那么就应该使用remove()方法：

```
mongo.remove(order);
```

如我前面所述，MongoOperations有多个操作文档数据的方法。我建议你查看一下其JavaDoc文档，以了解通过MongoOperations都能完成什么功能。

通常来讲，我们会将MongoOperations注入到自己设计的Repository类中，并使用它的操作来实现Repository方法。但是，如果你不愿意编写Repository的话，那么Spring Data MongoDB能够自动在运行时生成Repository实现。下面，我们来看一下是如何实现的。

12.1.4 编写MongoDB Repository

为了理解如何使用Spring Data MongoDB来创建Repository，让我们先回忆一下在第11章中是如何使用Spring Data JPA的。在程序清单11.4中，我们创建了一个扩展自JpaRepository的SpitterRepository接口。在那一小节中，我们还启用了Spring Data JPA Repository功能。这样的结果就是Spring Data JPA能够自动创建接口的实现，其中包括了多个内置的方法以及我们所添加的遵循命名约定的方法。

我们已经通过@EnableMongoRepositories注解启用了Spring Data MongoDB的Repository功能，接下来需要做的就是创建一个接口，Repository实现要基于这个接口来生成。不过，在这里，我们不再扩展JpaRepository，而是要扩展MongoRepository。如下程序清单中的OrderRepository扩展了MongoRepository，为Order文档提供了基本的CRUD操作。

程序清单**12.6　Spring Data MongoDB**会自动实现**Repository**接口

```
package orders.db;
import orders.Order;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface OrderRepository
        extends MongoRepository<Order, String> {
}
```

因为OrderRepository扩展了MongoRepository，因此它就会传递性地扩展Repository标记接口。回忆一下我们在学习Spring Data JPA时所了解的知识，任何扩展Repository的接口将会在运行时自动生成实现。在本例中，并不会实现与关系型数据库交互的JPA

Repository，而是会为OrderRepository生成读取和写入数据到MongoDB文档数据库的实现。

MongoRepository接口有两个参数，第一个是带有@Document注解的对象类型，也就是该Repository要处理的类型。第二个参数是带有@Id注解的属性类型。

尽管OrderRepository本身并没有定义任何方法，但是它会继承多个方法，包括对Order文档进行CRUD操作的方法。表12.2描述了OrderRepository继承的所有方法。

表**12.2**　通过扩展**MongoRepository**，**Repository**接口能够继承多个**CRUD**操作，它们会由**Spring Data MongoDB**自动实现

方　　法	描　　述
`long count();`	返回指定Repository类型的文档数量
`void delete(Iterable<? extends T);`	删除与指定对象关联的所有文档
`void delete(T);`	删除与指定对象关联的文档
`void delete(ID);`	根据ID删除某一个文档
`void deleteAll();`	删除指定Repository类型的所有文档
`boolean exists(Object);`	如果存在与指定对象相关联的文档，则返回true
`boolean exists(ID);`	如果存在指定ID的文档，则返回true
`List<T> findAll();`	返回指定Repository类型的所有文档
`List<T> findAll(Iterable<ID>);`	返回指定文档ID对应的所有文档
	为指定的Repository类型，返回分页且排序的文

	档列表
List<T> findAll(Sort);	为指定的Repository类型，返回排序后的所有文档列表
T findOne(ID);	为指定的ID返回单个文档
Save(terable<s>) ;	保存指定Iterable中的所有文档
save (< S >);	为给定的对象保存一条文档

表12.2中的方法使用了传递进来和方法返回的泛型。OrderRepository扩展了MongoRepository<Order, String>，那么T就映射为Order，ID映射为String，而S映射为所有扩展Order的类型。

添加自定义的查询方法

通常来讲，CRUD操作是很有用的，但我们有时候可能希望Repository提供除内置方法以外的其他方法。

在11.3.1小节中，我们学习了Spring Data JPA支持方法命名约定，它能够帮助Spring Data为遵循约定的方法自动生成实现。实际上，相同的约定也适用于Spring Data MongoDB。这意味着我们可以为OrderRepository添加自定义的方法：

```
public interface OrderRepository
        extends MongoRepository<Order, String> {
  List<Order> findByCustomer(String c);
  List<Order> findByCustomerLike(String c);
  List<Order> findByCustomerAndType(String c, String t);
  List<Order> findByCustomerLikeAndType(String c, String t);
}
```

这里我们有四个新的方法，每一个都是查找满足特定条件的Order对象。其中第一个用来获取customer属性等于传入值的Order列表；第二个方法获取customer属性like传入值的Order列表；接下来方法会返回customer和type属性等于传入值的Order对象；最后一个方法与前一个类似，只不过customer在对比的时候使用的是*like*而不是*equals*。

其中，find这个查询动词并不是固定的。如果喜欢的话，我们还可以使用get作为查询动词：

```
List<Order> getByCustomer(String c);
```

如果read更适合的话，你还可以使用这个动词：

```
List<Order> readByCustomer(String c);
```

除此之外，还有一个特殊的动词用来为匹配的对象计数：

```
int countByCustomer(String c);
```

与Spring Data JPA类似，在查询动词与By之前，我们有很大的灵活性。例如，我们可以标示要查找什么内容：

```
List<Order> findOrdersByCustomer(String c);
```

其中，Orders这个词没并没有什么特殊之处，它不会影响要获取的内容。我们也可以将方法按照如下的方式命名：

```
List<Order> findSomeStuffWeNeedByCustomer(String c);
```

其实，并不是必须要返回List<Order>，如果只想要一个Order对象的话，我们可以只需简单地返回Order：

```
Order findASingleOrderByCustomer(String c);
```

这里，所返回的就是原本List中的第一个Order对象。如果没有匹配元素的话，方法将会返回null。

指定查询

在11.3.2小节中，@Query注解可以为Repository方法指定自定义的查询。@Query能够像在JPA中那样用在MongoDB上。唯一的区别在于针对MongoDB时，@Query会接受一个JSON查询，而不是JPA查询。

例如，假设我们想要查询给定类型的订单，并且要求customer的名称为“Chuck Wagon”。OrderRepository中如下的方法声明能够完成所需的任务：

```
@Query("{'customer': 'Chuck Wagon', 'type' : ?0}")
List<Order> findChucksOrders(String t);
```

@Query中给定的JSON将会与所有的Order文档进行匹配，并返回匹配的文档。需要注意的是，type属性映射成了“?0”，这表明type属性应该与查询方法的第零个参数相等。如果有多个参数的话，它们可以通过“?1”、“?2”等方式进行引用。

混合自定义的功能

在11.3.3小节中，我们学习了如何将完全自定义的方法混合到自动生成的Repository中。对于JPA来说，这还涉及到创建一个中间接口来声明自定义的方法，为这些自定义方法创建实现类并修改自动化的Repository接口，使其扩展中间接口。对于Spring Data MongoDB来说，这些步骤都是相同的。

假设我们想要查询文档中type属性匹配给定值的Order对象。我们可以通过创建签名为List<Order> findByType(String t)的方法，很容易实现这个功能。但是，如果给定的类型是“NET”，那我们就查找type值为“WEB”的Order对象。要实现这个功能的话，这就有些困难了，即便使用@Query注解也不容易实现。不过，混合实现的做法能够完成这项任务。

首先，定义中间接口：

```
package orders.db;
import java.util.List;
import orders.Order;

public interface OrderOperations {
  List<Order> findOrdersByType(String t);
}
```

这非常简单。接下来，我们要编写混合实现，具体实现如下面的程序清单所示。

程序清单**12.7**　将自定义的**Repository**功能注入到自动生成的**Repository**中

```
package orders.db;
import java.util.List;
import orders.Order;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;

public class OrderRepositoryImpl implements OrderOperations {
  @Autowired
  private MongoOperations mongo;                    <-- 注入 MongoOperations

  public List<Order> findOrdersByType(String t) {
    String type = t.equals("NET") ? "WEB" : t;

    Criteria where = Criteria.where("type").is(t);          <-- 创建查询
    Query query = Query.query(where);

    return mongo.find(query, Order.class);          <-- 执行查询
  }
}
```

可以看到，混合实现中注入了MongoOperations（也就是MongoTemplate所实现的接口）。findOrdersByType()方法使用MongoOperations对数据库进行了查询，查找匹配条件的文档。

剩下的工作就是修改OrderRepository，让其扩展中间接口OrderOperations：

```
public interface OrderRepository
       extends MongoRepository<Order, String>, OrderOperations {
  ...
}
```

将这些关联起来的关键点在于实现类的名称为OrderRepositoryImpl。这个名字前半部分与OrderRepository相同，只是添加了“Impl”后缀。当Spring Data MongoDB生成Repository实现时，它会查找这个类并将其混合到自动生成的实现中。

如果你不喜欢“Impl”后缀的话，那么可以配置Spring Data MongoDB，让其按照名字查找具备不同后缀的类。我们需要做的就是设置@EnableMongoRepositories的属性（在Spring配置类中）：

```
@Configuration
@EnableMongoRepositories(basePackages="orders.db",
                         repositoryImplementationPostfix="Stuff")
public class MongoConfig extends AbstractMongoConfiguration {
  ...
}
```

如果使用XML配置的话，我们可以设置<mongo:repositories>的repository-impl-postfix属性：

```
<mongo:repositories base-package="orders.db"
                    repository-impl-postfix="Stuff" />
```

不管采用哪种方式，我们现在都让Spring Data MongoDB查找名为`OrderRepositoryStuff`的类，而不再查找`OrderRepositoryImpl`。

像MongoDB这样的文档数据库能够解决特定类型的问题，但是就像关系型数据库不是全能型数据库那样，MongoDB同样如此。有些问题并不是关系型数据库或文档型数据库适合解决的，不过，幸好我们的选择并不仅限于这两种。

接下来，我们看一下Spring Data如何支持Neo4j，这是一种很流行的图数据库。

12.2 使用Neo4j操作图数据

文档型数据库会将数据存储到粗粒度的文档中，而图数据库会将数据存储到多个细粒度的节点中，这些节点之间通过关系建立关联。图数据库中的一个节点通常会对应数据库中的一个概念（concept），它会具备描述节点状态的属性。连接两个节点的关联关系可能也会带有属性。

按照其最简单的形式，图数据库比文档数据库更加通用，有可能会成为关系型数据库的无模式（schemaless）替代方案。因为数据的结构是图，所以可以遍历关联关系以查找数据中你所关心的内容，这在其他数据库中是很难甚至无法实现的。

Spring Data Neo4j提供了很多与Spring Data JPA和Spring Data MongoDB相同的功能，当然所针对的是Neo4j图数据库。它提供了将Java对象映射到节点和关联关系的注解、面向模板的Neo4j访问方式以及Repository实现的自动化生成功能。

我们稍后会看到如何在Neo4j中使用这些特性，不过首先我们需要配置Spring Data Neo4j。

12.2.1 配置Spring Data Neo4j

配置Spring Data Neo4j的关键在于声明`GraphDatabaseService` bean和启用Neo4j Repository自动生成功能。如下的程序清单展现了Spring Data Neo4j所需的基本配置。

程序清单12.8　使用@EnableNeo4jRepositories来配置Spring Data Neo4j

```
package orders.config;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.neo4j.config.EnableNeo4jRepositories;
import org.springframework.data.neo4j.config.Neo4jConfiguration;
@Configuration
@EnableNeo4jRepositories(basePackages="orders.db")    <-- 启用Repository自动生成功能
public class Neo4jConfig extends Neo4jConfiguration {

  public Neo4jConfig() {
    setBasePackage("orders");    <-- 设置模型的基础包
  }

  @Bean(destroyMethod="shutdown")
  public GraphDatabaseService graphDatabaseService() {
    return new GraphDatabaseFactory()
          .newEmbeddedDatabase("/tmp/graphdb");    <-- 配置嵌入式数据库
  }

}
```

@EnableNeo4jRepositories注解能够让Spring Data Neo4j自动生成Neo4j Repository实现。它的basePackages属性设置为orders.db包，这样它就会扫描这个包来查找（直接或间接）扩展Repository标记接口的其他接口。

Neo4jConfig扩展自Neo4jConfiguration，后者提供了多个便利的方法来配置Spring Data Neo4j。在这些方法中，就包括setBasePackage()，它会在Neo4jConfig的构造器中调用，用来告诉Spring Data Neo4j要在orders包中查找模型类。

这个拼图的最后一部分是定义GraphDatabaseServicebean。在本例中，graphDatabaseService()方法使用GraphDatabaseFactory来创建嵌入式的Neo4j数据库。在Neo4j中，嵌入式数据库不要与内存数据库相混淆。在这里，“嵌入式”指的是数据库引擎与应用运行在同一个JVM中，作为应用的一部分，而不是独立的服务器。数据依然会持久化到文件系统中（在本例中，也就是“/tmp/graphdb”中）。

作为另外的一种方式，你可能会希望配置GraphDatabaseService连接远程的Neo4j服务器。如果spring-data-neo4j-rest库在应用的类路径下，那么我们就可以配置SpringRestGraphDatabase，它会通过RESTful API来访问远程的Neo4j数据库：

```
@Bean(destroyMethod="shutdown")
public GraphDatabaseService graphDatabaseService() {
  return new SpringRestGraphDatabase(
      "http://graphdbserver:7474/db/data/");
}
```

如上所示，SpringRestGraphDatabase在配置时，假设远程的数据库并不需要认证。但是，在生产环境的配置中，当创建SpringRestGraphDatabase的时候，我们可能希望提供应用的凭证：

```
@Bean(destroyMethod="shutdown")
public GraphDatabaseService graphDatabaseService(Environment env) {
  return new SpringRestGraphDatabase(
      "http://graphdbserver:7474/db/data/",
      env.getProperty("db.username"), env.getProperty("db.password"));
}
```

在这里，凭证是通过注入的Environment获取到的，避免了在配置类中的硬编码。

Spring Data Neo4j同时还提供了XML命名空间。如果你更愿意在XML中配置Spring Data Neo4j的话，那可以使用该命名空间中的<neo4j:config>和<neo4j:repositories>元素。在功能上，程序清单12.9所展示的配置与程序清单12.8中的Java配置是相同的。

程序清单**12.9　Spring Data Neo4j**也可以通过**XML**来配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
  xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/data/neo4j
   http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">
  <neo4j:config                                      <-- 配置 Neo4j
      storeDirectory="/tmp/graphdb"                      数据库的细节
      base-package="orders" />

  <neo4j:repositories base-package="orders.db" />  <-- 启用 Repository 生成功能
</beans>
```

<neo4j:config>元素配置了如何访问数据库的细节。在本例中，它配置Spring Data Neo4j使用嵌入式的数据库。具体来讲，storeDirectory属性指定了数据要持久化到哪个文件系统路径中。base-package属性设置了模型类定义在哪个包中。

至于<neo4j:repositories>元素，它启用Spring Data Neo4j自动生成Repository实现的功能，它会扫描orders.db包，查找所有扩

展Repository的接口。

如果要配置Spring Data Neo4j访问远程的Neo4j服务器，我们所需要做的就是声明SpringRestGraphDatabasebean，并设置<neo4j:config>的graphDatabaseService属性：

```
<neo4j:config base-package="orders"
              graphDatabaseService="graphDatabaseService" />

<bean id="graphDatabaseService" class=
        "org.springframework.data.neo4j.rest.SpringRestGraphDatabase">
  <constructor-arg value="http://graphdbserver:7474/db/data/" />
  <constructor-arg value="db.username" />
  <constructor-arg value="db.password" />
</bean>
```

不管是通过Java还是通过XML来配置Spring Data Neo4j，我们都需要确保模型类位于基础包所指定的包中（通过@EnableNeo4jRepositories的basePackages属性或<neo4j:config>的base-package属性来进行设置）。它们都需要使用注解将其标注为节点实体或关联关系实体。这就是我们接下来的任务。

12.2.2　使用注解标注图实体

Neo4j定义了两种类型的实体：节点（node）和关联关系（relationship）。一般来讲，节点反映了应用中的事物，而关联关系定义了这些事物是如何联系在一起的。

Spring Data Neo4j提供了多个注解，它们可以应用在模型类型及其域上，实现Neo4j中的持久化。表12.3描述了这些注解。

表**12.3**　借助**Spring Data Neo4j**的注解，能够将领域类型映射为图中的节点和关联关系

注　　解	描　　述
@NodeEntity	将Java类型声明为节点实体
@RelationshipEntity	将Java类型声明为关联关系实体
@StartNode	将某个属性声明为关联关系实体的开始节点

@EndNode	将某个属性声明为关联关系实体的结束节点
@Fetch	将实体的属性声明为立即加载
@GraphId	将某个属性设置为实体的ID域（这个域的类型必须是Long）
@GraphProperty	明确声明某个属性
@GraphTraversal	声明某个属性会自动提供一个iterable元素，这个元素是图遍历所构建的
@Indexed	声明某个属性应该被索引
@Labels	为@NodeEntity声明标签
@Query	声明某个属性会自动提供一个iterable元素，这个元素是执行给定的Cypher查询所构建的
@QueryResult	声明某个Java或接口能够持有查询的结果
@RelatedTo	通过某个属性，声明当前的@NodeEntity与另外一个@NodeEntity之间的关联关系
@RelatedToVia	在@NodeEntity上声明某个属性，指定其引用该节点所属的某一个@RelationshipEntity
@RelationshipType	将某个域声明为关联实体类型
@ResultColumn	在带有@QueryResult注解的类型上，将某个属性声明为获取查询结果集中的某个特定列

为了了解如何使用其中的某些注解，我们会将其应用到订单/条目样例中。

在该样例中，数据建模的一种方式就是将订单设定为一个节点，它会与一个或多个条目关联。图12.2以图的形式描述了这种模型。

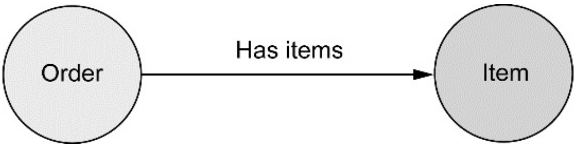

图12.2　连接两个节点的简单关联关系，关系本身不包含任何属性

为了将订单指定为节点，我们需要为Order类添加@NodeEntity注解。如下的程序清单展现了带有@NodeEntity注解的Order类，它还包含了表12.3中的几个其他注解。

程序清单**12.10**　为**Order**添加注解，使其成为图数据库中的一个节点

```
package orders;
import java.util.LinkedHashSet;
import java.util.Set;
import org.springframework.data.neo4j.annotation.GraphId;
import org.springframework.data.neo4j.annotation.NodeEntity;
import org.springframework.data.neo4j.annotation.RelatedTo;

@NodeEntity                                    <— Order 类是节点
public class Order {

  @GraphId                                     <— Graph ID
  private Long id;
  private String customer;
  private String type;

  @RelatedTo(type="HAS_ITEMS")                 <— 与条目之间的关联关系
  private Set<Item> items = new LinkedHashSet<Item>();

  ...

}
```

除了类级别上的@NodeEntity，还要注意id属性上使用了@GraphId注解。Neo4j上的所有实体必要要有一个图ID。这大致类似于JPA @Entity以及MongoDB @Document类中使用@Id注解的属性。在这里，@GraphId注解标注的属性必须是Long类型。

customer和type属性上没有任何注解。只要这些属性不是瞬态的，它们都会成为数据库中节点的属性。

items属性上使用了@RelatedTo注解，这表明Order与一个Item的Set存在关联关系。type属性实际上就是为关联关系建立了一个文本标记。它可以设置成任意的值，但通常会给定一个易于人类阅读的文本，用来简单描述这个关联关系的特征。稍后，你将会看到如何

将这个标记用在查询中，实现跨关联关系的查询。

就Item本身来说，下面展现了如何为其添加注解实现图的持久化。

程序清单**12.11　Item**也是图数据库中的节点

```
package orders;
import org.springframework.data.neo4j.annotation.GraphId;
import org.springframework.data.neo4j.annotation.NodeEntity;

@NodeEntity                                    <— Item 类是节点
public class Item {

  @GraphId                                     <— Graph ID
  private Long id;
  private String product;
  private double price;
  private int quantity;

  ...

}
```

类似于Order，Item也使用了@NodeEntity注解，将其标记为一个节点。它同时也有一个Long类型的属性，借助@GraphId注解将其标注为节点的图ID，而product、price以及quantity属性均会作为图数据库中节点的属性。

Order和Item之间的关联关系很简单，关系本身并不包含任何的数据。因此，@RelatedTo注解就足以定义关联关系。但是，并不是所有的关联关系都这么简单。

让我们重新考虑该如何为数据建模，从而学习如何使用更为复杂的关联关系。在当前的数据模型中，我们将条目和产品的信息组合到了Item类中。但是，当我们重新考虑的时候，会发现订单会与一个或多个产品相关联。订单与产品之间的关系构成了订单的一个条目。图12.3描述了另外一种在图中建模数据的方式。

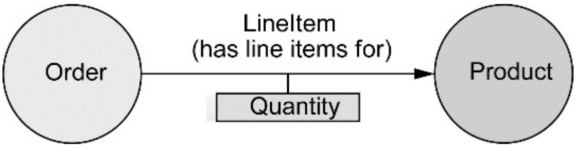

图12.3　关联关系实体自身具有属性

在这个新的模型中，订单中产品的数量是条目中的一个属性，而产品本身是另外一个概念。与前面一样，订单和产品都是节点，而条目是关联关系。因为现在的条目必须要包含一个数量值，关联关系不像前面那么简单。我们需要定义一个类来代表条目，比如如下程序清单所示的LineItem。

程序清单12.12　**LineItem**类连接了一个**Order**节点和一个**Product**节点

```
package orders;
import org.springframework.data.neo4j.annotation.EndNode;
import org.springframework.data.neo4j.annotation.GraphId;
import org.springframework.data.neo4j.annotation.RelationshipEntity;
import org.springframework.data.neo4j.annotation.StartNode;

@RelationshipEntity(type="HAS_LINE_ITEM_FOR")    <-- LineItem 是关联关系
public class LineItem {

  @GraphId                          <-- Graph ID
  private Long id;

  @StartNode                          <-- 开始节点
  private Order order;

  @EndNode                             <-- 结束节点
  private Product product;

  private int quantity;

  ...

}
```

Order类通过`@NodeEntity`注解将其标示为一个节点，而`LineItem`类则使用了`@RelationshipEntity`注解。`LineItem`同样也有一个`id`属性标注了`@GraphId`注解，不管是节点实体还是关联关系实体，都必须要有一个图ID，而且其类型必须为`Long`。

关联关系实体的特殊之处在于它们连接了两个节点。`@StartNode`和`@EndNode`注解用在定义关联关系两端的属性上。在本例中，`Order`是开始节点，`Product`是结束节点。

最后，`LineItem`类有一个`quantity`属性，当关联关系创建的时候，它会持久化到数据库中。

领域对象已经添加了注解，现在就可以保存与读取节点和关联关系了。我们首先看一下如何使用Spring Data Neo4j中的`Neo4jTemplate`实现面向模板的数据访问。

12.2.3　使用Neo4jTemplate

Spring Data MongoDB提供了`MongoTemplate`实现基于模板的MongoDB持久化，与之类似，Spring Data Neo4j提供了`Neo4jTemplate`来操作Neo4j图数据库中的节点和关联关系。如果你已经按照前面的方式配置了Spring Data Neo4j，在Spring应用上下文中就已经具备了一个`Neo4jTemplate`bean。接下来需要做的就是将其注入到任意想使用它的地方。

例如，我们可以直接将其自动装配到某个bean的属性上：

```
@Autowired
private Neo4jOperations neo4j;
```

Neo4jTemplate定义了很多的方法，包括保存节点、删除节点以及创建节点间的关联关系。我们没有足够的篇幅介绍所有的方法，但是我们会看一下Neo4jTemplate所提供的最为常用的方法。

我们想借助Neo4jTemplate完成的最基本的一件事情可能就是将某个对象保存为节点。假设这个对象已经使用了@NodeEntity注解，那么我们可以按照如下的方式来使用save()方法：

```
Order order = ...;
Order savedOrder = neo4j.save(order);
```

如果你能知道对象的图ID，那么可以通过findOne()方法来获取它：

```
Order order = neo4j.findOne(42, Order.class);
```

如果按照给定的ID找不到节点的话，那么findOne()方法将会抛出NotFound(Exception)。

如果你想获取给定类型的所有对象，那么可以使用findAll()方法：

```
EndResult<Order> allOrders = neo4j.findAll(Order.class);
```

这里返回的EndResult是一个Iterable，它能够用在for-each循环以及任何可以使用Iterable的地方。如果不存在这样的节点的话，findAll()方法将会返回空的Iterable。

如果你只是想知道Neo4j数据库中指定类型的对象数量，那么就可以调用count()方法：

```
long orderCount = count(Order.class);
```

delete()方法可以用来删除对象：

```
neo4j.delete(order);
```

createRelationshipBetween()是Neo4jTemplate所提供的最有意思的方法之一。我们可以猜到，它会为两个节点创建关联关系。

例如，我们可以在Order节点和Product节点之间建立LineItem关联关系：

```
Order order = ...;
Product prod = ...;
LineItem lineItem = neo4j.createRelationshipBetween(
    order, prod, LineItem.class, "HAS_LINE_ITEM_FOR", false);
lineItem.setQuantity(5);
neo4j.save(lineItem);
```

createRelationshipBetween()方法的前两个参数是关联关系两端的节点对象。接下来的参数指定了使用@RelationshipEntity注解的类型，它会代表这种关系。接下来的String值描述了关联关系的特征。最后的参数是一个boolean值，它表明这两个节点实体之间是否允许存在重复的关联关系。

createRelationshipBetween()会返回关联关系类的一个实例。通过它，我们可以设置任意的属性。上面的示例中设置了quantity属性。当这一切完成后，我们调用save()方法将关联关系保存到数据库中。

Neo4jTemplate提供了很便利的方式来使用Neo4j图数据库中的节点和关联关系。但是，这种方式需要借助Neo4jTemplate编写自己的Repository实现。接下来，我们看一下Spring Data Neo4j怎样为我们自动化生成Repository实现。

12.2.4 创建自动化的Neo4j Repository

大多数Spring Data项目都具备的最棒的一项功能就是为Repository接口自动生成实现。我们已经在Spring Data JPA和Spring Data MongoDB中看到了这项功能。Spring Data Neo4j也不例外，它同样支持Repository自动化生成功能。

我们已经将@EnableNeo4jRepositories添加到了配置中，所以Spring Data Neo4j已经配置为支持自动化生成Repository的功能。我们所需要做的就是编写接口，如下的OrderRepository就是很好的起点：

```
package orders.db;
import orders.Order;
import org.springframework.data.neo4j.repository.GraphRepository;

public interface OrderRepository extends GraphRepository<Order> {}
```

与其他的Spring Data项目一样，Spring Data Neo4j会为扩展Repository

接口的其他接口生成Repository方法实现。在本例中，OrderRepository扩展了GraphRepository，而后者又间接扩展了Repository接口。因此，Spring Data Neo4j将会在运行时创建OrderRepository的实现。

注意，GraphRepository使用Order进行了参数化，也就是这个Repository所要使用的实体类型。因为Neo4j要求图ID的类型为Long，因此在扩展GraphRepository的时候，没有必要再去指定ID类型。

现在，我们就能够使用很多通用的CRUD操作，这与JpaRepository和MongoRepository所提供的功能类似。表12.4描述了扩展GraphRepository所能够得到的方法。

表**12.4**　通过扩展**GraphRepository**，**Repository**接口能够继承多个**CRUD**操作，
它们会由**Spring Data Neo4j**自动实现

方　　法	描　　述
`long count();`	返回在数据库中，目标类型有多少实体
`void delete(Iterable<?extendsT>);`	删除多个实体
`void delete(Long id);`	根据ID，删除一个实体
`void delete(T);`	删除一个实体
`void deleteAll();`	删除目标类型的所有实体
`boolean exists(Long id);`	根据指定的ID，检查实体是否存在
`EndResult<T> findAll();`	获取目标类型的所有实体
	根据给定的ID，获取目标类型的

	实体
`Page<T> findAll(Pageable);`	返回目标类型分页和排序后的实体列表
`EndResult<T> findAll(Sort);`	返回目标类型排序后的实体列表
`EndResult<T> findAllBySchemaPropertyValue(String,Object);`	返回指定属性匹配给定值的所有实体
`Iterable<T> findAllByTraversal(N, TraversalDescription);`	返回从某个节点开始，图遍历到达的节点
`T findBySchemaPropertyValue (String,Object);`	返回指定属性匹配给定值的一个实体
`T findOne(Long);`	根据ID，获得某一个实体
`EndResult<T> query(String, Map<String,Object>);`	返回匹配给定Cypher查询的所有实体
`Iterable<T> save(Iterable<T>);`	保存多个实体
`S save(S);`	保存一个实体

我们没有足够的篇幅介绍所有的方法，但是有些方法你可能会经常用到。例如，如下的代码能够保存一个`Order`实体：

```
Order savedOrder = orderRepository.save(order);
```

当实体保存之后，`save()`方法将会返回被保存的实体，如果之前它使用`@GraphId`注解的属性值为`null`的话，此时这个属性将会填充上值。

我们还可以使用`findOne()`方法查询某一个实体。例如，下面的这行代码将会查询图ID为4的`Order`：

行代码将会查询图ID为4的`Order`：

```
Order order = orderRepository.findOne(4L);
```

我们还可以查询所有的Order：

```
EndResult<Order> allOrders = orderRepository.findAll();
```

当然，你可能还希望删除某一个实体。这种情况下，可以使用`delete()`方法：

```
delete(order);
```

这将会从数据库中删除给定的`Order`节点。如果你只有图ID的话，那可以将其传递到`delete()`方法中，而不是再使用节点类型本身：

```
delete(orderId);
```

如果你希望进行自定义的查询，那么可以使用`query()`方法对数据库执行任意的Cypher查询。但是这与使用`Neo4jTemplate`的`query()`方法并没有太大的差别。其实，我们还可以为`OrderRepository`添加自定义的查询方法。

添加查询方法

我们已经看过如何按照命名约定使用Spring Data JPA和Spring Data MongoDB来添加自定义的查询方法。如果Spring Data Neo4j没有提供相同功能的话，那我们就该失望了。

如下面的程序清单所示，其实我们完全没有必要失望：

程序清单**12.13**　通过遵循命名约定来定义查询方法

```
package orders.db;
import java.util.List;
import orders.Order;

import org.springframework.data.neo4j.repository.GraphRepository;

public interface OrderRepository extends GraphRepository<Order> {
                                                                          查询方法
  List<Order> findByCustomer(String customer);                        <─┘
  List<Order> findByCustomerAndType(String customer, String type);

}
```

这里，我们添加了两个方法。其中一个会查询`customer`属性等于给定`String`值的`Order`节点。另外一个方法与之类似，但是除了匹

我们之前已经讨论过查询方法的命名约定，所以这里没有必要再进行深入地讨论。可以翻看之前学习Spring Data JPA的章节，重新温习如何编写这些方法。

指定自定义查询

当命名约定无法满足需求时，我们还可以为方法添加@Query注解，为其指定自定义的查询。我们之前已经见过@Query注解。在Spring Data JPA中，我们使用它来为Repository方法指定JPA查询。在Spring Data MongoDB中，我们使用它来指定匹配JSON的查询。但是，在使用Spring Data Neo4j的时候，我们必须指定Cypher查询：

```
@Query("match (o:Order)-[:HAS_ITEMS]->(i:Item) " +
       "where i.product='Spring in Action' return o")
List<Order> findSiAOrders();
```

在这里，findSiAOrders()方法上使用了@Query注解，并设置了一个Cypher查询，它会查找与Item关联并且product属性等于“Spring in Action”的所有Order节点。

混合自定义的Repository行为

当命名约定和@Query注解均无法满足满足需求的时候，我们还可以混合自定义的Repository逻辑。

例如，假设我们想自己编写findSiAOrders()方法的实现，而不是依赖于@Query注解。那么可以首先定义一个中间接口，该接口包含findSiAOrders()方法的定义：

```
package orders.db;
import java.util.List;
import orders.Order;

public interface OrderOperations {
  List<Order> findSiAOrders();
}
```

然后，我们修改OrderRepository，让它扩展OrderOperations和GraphRepository：

```
public interface OrderRepository
       extends GraphRepository<Order>, OrderOperations {

    ...

}
```

最后，我们需要自己编写实现。与Spring Data JPA和Spring Data

```
public interface OrderRepository
       extends GraphRepository<Order>, OrderOperations {
   ...
}
```

最后，我们需要自己编写实现。与Spring Data JPA和Spring Data MongoDB类似，Spring Data Neo4j将会查找名字与Repository接口相同且添加“Impl”后缀的实现类。因此，我们需要创建OrderRepositoryImpl类。如下的程序清单展示了OrderRepositoryImpl类，它实现了findSiAOrders()方法。

程序清单**12.14**　将自定义功能混合到**OrderRepository**中

```
package orders.db;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import orders.Order;
import org.neo4j.helpers.collection.IteratorUtil;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.neo4j.conversion.EndResult;
import org.springframework.data.neo4j.conversion.Result;
import org.springframework.data.neo4j.template.Neo4jOperations;

public class OrderRepositoryImpl implements OrderOperations {    <-- 实现中间接口

  private final Neo4jOperations neo4j;

  @Autowired
  public OrderRepositoryImpl(Neo4jOperations neo4j) {    <-- 注入Neo4jOperations
    this.neo4j = neo4j;
  }

  public List<Order> findSiAOrders() {
    Result<Map<String, Object>> result = neo4j.query(    <-- 执行查询
           "match (o:Order)-[:HAS_ITEMS]->(i:Item) " +
           "where i.product='Spring in Action' return o",

   EndResult<Order> endResult = result.to(Order.class);    <-- 转换为EndResult<Order>

   return IteratorUtil.asList(endResult);    <-- 转换为List<Order>
  }
}
```

OrderRepositoryImpl中注入了一个Neo4jOperations（具体来讲，就是Neo4jTemplate的实例），它会用来查询数据库。因为query()方法返回的是Result<Map<String, Object>>，我们需要将其转换为List<Order>。第一步是调用Result的to()方法，产生一个EndResult<Order>。然后，使用Neo4j的IteratorUtil.asList()方法将EndResult<Order>转换为List<Order>，然后将其返回。

对于能够表达为节点和关联关系的数据，像Neo4j这样的图数据库是非常合适的。如果将我们生活的世界理解为各种互相关联的事物，那么图数据库能够适用于很大的范围。就我个人而言，我非常喜欢

的数据持久化。

12.3 使用Redis操作key-value数据

Redis是一种特殊类型的数据库，它被称之为key-value存储。顾名思义，key-value存储保存的是键值对。实际上，key-value存储与哈希Map有很大的相似性。可以不太夸张地说，它们就是持久化的哈希Map。

当你思考这一点的时候，可能会意识到，对于哈希Map或者key-value存储来说，其实并没有太多的操作。我们可以将某个value存储到特定的key上，并且能够根据特定key，获取value。差不多也就是这样了。因此，Spring Data的自动Repository生成功能并没有应用到Redis上。不过，Spring Data的另外一个关键特性，也就是面向模板的数据访问，能够在使用Redis的时候，为我们提供帮助。

Spring Data Redis包含了多个模板实现，用来完成Redis数据库的数据存取功能。稍后，我们就会看到如何使用它们。但是为了创建Spring Data Redis的模板，我们首先需要有一个Redis连接工厂。幸好，Spring Data Redis提供了四个连接工厂供我们选择。

12.3.1 连接到Redis

Redis连接工厂会生成到Redis数据库服务器的连接。Spring Data Redis为四种Redis客户端实现提供了连接工厂：

- `JedisConnectionFactory`
- `JredisConnectionFactory`
- `LettuceConnectionFactory`
- `SrpConnectionFactory`

具体选择哪一个取决于你。我建议你自行测试并建立基准，进而确定哪一种Redis客户端和连接工厂最适合你的需求。从Spring Data Redis的角度来看，这些连接工厂在适用性上都是相同的。

在做出决策之后，我们就可以将连接工厂配置为Spring中的bean。例如，如下展示了如何配置`JedisConnectionFactory` bean：

在做出决策之后，我们就可以将连接工厂配置为Spring中的bean。例如，如下展示了如何配置JedisConnectionFactory bean：

```
@Bean
public RedisConnectionFactory redisCF() {
  return new JedisConnectionFactory();
}
```

通过默认构造器创建的连接工厂会向localhost上的6379端口创建连接，并且没有密码。如果你的Redis服务器运行在其他的主机或端口上，在创建连接工厂的时候，可以设置这些属性：

```
@Bean
public RedisConnectionFactory redisCF() {
  JedisConnectionFactory cf = new JedisConnectionFactory();
  cf.setHostName("redis-server");
  cf.setPort(7379);
  return cf;
}
```

类似地，如果你的Redis服务器配置为需要客户端认证的话，那么可以通过调用setPassword()方法来设置密码：

```
@Bean
public RedisConnectionFactory redisCF() {
  JedisConnectionFactory cf = new JedisConnectionFactory();
  cf.setHostName("redis-server");
  cf.setPort(7379);
  cf.setPassword("foobared");
  return cf;
}
```

在上面的这些例子中，我都假设使用的是JedisConnectionFactory。如果你选择使用其他连接工厂的话，只需进行简单地替换就可以了。例如，假设你要使用LettuceConnectionFactory的话，可以按照如下的方式进行配置：

```
@Bean
public RedisConnectionFactory redisCF() {
  JedisConnectionFactory cf = new LettuceConnectionFactory();
  cf.setHostName("redis-server");
  cf.setPort(7379);
  cf.setPassword("foobared");
  return cf;
}
```

所有的Redis连接工厂都具有setHostName()、setPort()和setPassword()方法。这样，它们在配置方面实际上是相同的。

现在，我们有了Redis连接工厂，接下来就可以使用Spring Data Redis模板了。

顾名思义，Redis连接工厂会生成到Redis key-value存储的连接（以RedisConnection的形式）。借助RedisConnection，可以存储和读取数据。例如，我们可以获取连接并使用它来保存一个问候信息，如下所示：

```
RedisConnectionFactory cf = ...;
RedisConnection conn = cf.getConnection();
conn.set("greeting".getBytes(), "Hello World".getBytes());
```

与之类似，我们还可以使用RedisConnection来获取之前存储的问候信息：

```
byte[] greetingBytes = conn.get("greeting".getBytes());
String greeting = new String(greetingBytes);
```

毫无疑问，这可以正常运行，但是你难道真的愿意使用字节数组吗？

与其他的Spring Data项目类似，Spring Data Redis以模板的形式提供了较高等级的数据访问方案。实际上，Spring Data Redis提供了两个模板：

- RedisTemplate
- StringRedisTemplate

RedisTemplate可以极大地简化Redis数据访问，能够让我们持久化各种类型的key和value，并不局限于字节数组。在认识到key和value通常是String类型之后，StringRedisTemplate扩展了RedisTemplate，只关注String类型。

假设我们已经有了RedisConnectionFactory，那么可以按照如下的方式构建RedisTemplate：

```
RedisConnectionFactory cf = ...;
RedisTemplate<String, Product> redis =
    new RedisTemplate<String, Product>();
redis.setConnectionFactory(cf);
```

注意，RedisTemplate使用两个类型进行了参数化。第一个是key的类型，第二个是value的类型。在这里所构建的RedisTemplate中，将会保存Product对象作为value，并将其赋予一个String类型的key。

如果你所使用的value和key都是String类型，那么可以考虑使用StringRedisTemplate来代替RedisTemplate：

用StringRedisTemplate来代替RedisTemplate：

```
RedisConnectionFactory cf = ...;
StringRedisTemplate redis = new StringRedisTemplate(cf);
```

注意，与RedisTemplate不同，StringRedisTemplate有一个接受RedisConnectionFactory的构造器，因此没有必要在构建后再调用setConnectionFactory()。

尽管这并非必须的，但是如果你经常使用RedisTemplate或StringRedisTemplate的话，你可以考虑将其配置为bean，然后注入到需要的地方。如下就是一个声明RedisTemplate的简单@Bean方法：

```
@Bean
public RedisTemplate<String, Product>
                            redisTemplate(RedisConnectionFactory cf) {
  RedisTemplate<String, Product> redis =
      new RedisTemplate<String, Product>();
  redis.setConnectionFactory(cf);
  return redis;
}
```

如下是声明StringRedisTemplate bean的@Bean方法：

```
@Bean
public StringRedisTemplate
                     stringRedisTemplate(RedisConnectionFactory cf) {
  return new StringRedisTemplate(cf);
}
```

有了RedisTemplate（或StringRedisTemplate）之后，我们就可以开始保存、获取以及删除key-value条目了。RedisTemplate的大多数操作都是表12.5中的子API提供的。

表**12.5　RedisTemplate**的很多功能是以子**API**的形式提供的，它们区分了单个值和集合值的场景

方　　法	子**API**接口	描　　述
opsForValue()	ValueOperations<K, V>	操作具有简单值的条目
opsForList()	ListOperations<K, V>	操作具有list值的条目
opsForSet()	SetOperations<K, V>	操作具有set值的条目

opsForZSet()	ZSetOperations<K, V>	操作具有ZSet值（排序的set）的条目
opsForHash()	HashOperations<K, HK, HV>	操作具有hash值的条目
boundValueOps(K)	BoundValueOperations<K,V>	以绑定指定key的方式，操作具有简单值的条目
boundListOps(K)	BoundListOperations<K,V>	以绑定指定key的方式，操作具有list值的条目
boundSetOps(K)	BoundSetOperations<K,V>	以绑定指定key的方式，操作具有set值的条目
boundZSet(K)	BoundZSetOperations<K,V>	以绑定指定key的方式，操作具有ZSet值（排序的set）的条目
boundHashOps(K)	BoundHashOperations<K,V>	以绑定指定key的方式，操作具有hash值的条目

我们可以看到，表12.5中的子API能够通过RedisTemplate（和StringRedis-Template）进行调用。其中每个子API都提供了使用数据条目的操作，基于value中所包含的是单个值还是一个值的集合它们会有所差别。

这些子API中，包含了很多从Redis中存取数据的方法。我们没有足够的篇幅介绍所有的方法，但是会介绍一些最为常用的操作。

使用简单的值

假设我们想通过RedisTemplate<String, Product>保存Product，其中key是sku属性的值。如下的代码片段展示了如何借助opsForValue()方法完成该功能：

```
redis.opsForValue().set(product.getSku(), product);
```

类似地，如果你希望获取sku属性为123456的产品，那么可以使用如下的代码片段：

如下的代码片段：

```
Product product = redis.opsForValue().get("123456");
```

如果按照给定的key，无法获得条目的话，将会返回null。

使用**List**类型的值

使用List类型的value与之类似，只需使用opsForList()方法即可。例如，我们可以在一个List类型的条目尾部添加一个值：

```
redis.opsForList().rightPush("cart", product)
```

通过这种方式，我们向列表的尾部添加了一个Product，所使用的这个列表在存储时key为cart。如果这个key尚未存在列表的话，将会创建一个。

rightPush()会在列表的尾部添加一个元素，而leftPush()则会在列表的头部添加一个值：

```
redis.opsForList().leftPush("cart", product);
```

我们有很多方式从列表中获取元素，可以通过leftPop()或rightPop()方法从列表中弹出一个元素：

```
Product first = redis.opsForList().leftPop("cart");
Product last = redis.opsForList().rightPop("cart");
```

除了从列表中获取值以外，这两个方法还有一个副作用就是从列表中移除所弹出的元素。如果你只是想获取值的话（甚至可能要在列表的中间获取），那么可以使用range()方法：

```
List<Product> products = redis.opsForList().range("cart", 2, 12);
```

range()方法不会从列表中移除任何元素，但是它会根据指定的key和索引范围，获取范围内的一个或多个值。前面的样例中，会获取11个元素，从索引为2的元素到索引为12的元素（不包含）。如果范围超出了列表的边界，那么只会返回索引在范围内的元素。如果该索引范围内没有元素的话，将会返回一个空的列表。

在**Set**上执行操作

除了操作列表以外，我们还可以使用opsForSet()操作Set。最为常

```
redis.opsForSet().add("cart", product);
```

在我们有多个Set并填充值之后，就可以对这些Set进行一些有意思的操作，如获取其差异、求交集和求并集：

```
List<Product> diff = redis.opsForSet().difference("cart1", "cart2");
List<Product> union = redis.opsForSet().union("cart1", "cart2");
List<Product> isect = redis.opsForSet().isect("cart1", "cart2");
```

当然，我们还可以移除它的元素：

```
redis.opsForSet().remove(product);
```

我们甚至还可以随机获取Set中的一个元素：

```
Product random = redis.opsForSet().randomMember("cart");
```

因为Set没有索引和内部的排序，因此我们无法精准定位某个点，然后从Set中获取元素。

绑定到某个**key**上

表12.5包含了五个子API，它们能够以绑定key的方式执行操作。这些子API与其他的API是对应的，但是关注于某一个给定的key。

为了举例阐述这些子API的用法，我们假设将`Product`对象保存到一个list中，并且key为`cart`。在这种场景下，假设我们想从list的右侧弹出一个元素，然后在list的尾部新增三个元素。我们此时可以使用`boundListOps()`方法所返回的`BoundListOperations`：

```
BoundListOperations<String, Product> cart =
        redis.boundListOps("cart");
Product popped = cart.rightPop();
cart.rightPush(product1);
cart.rightPush(product2);
cart.rightPush(product3);
```

注意，我们只在一个地方使用了条目的key，也就是调用`boundListOps()`的时候。对返回的`BoundListOperations`执行的所有操作都会应用到这个key上。

12.3.3　使用key和value的序列化器

当某个条目保存到Redis key-value存储的时候，key和value都会使用Redis的序列化器（serializer）进行序列化。Spring Data Redis提供了

当某个条目保存到Redis key-value存储的时候，key和value都会使用Redis的序列化器（serializer）进行序列化。Spring Data Redis提供了多个这样的序列化器，包括：

- GenericToStringSerializer：使用Spring转换服务进行序列化；
- JacksonJsonRedisSerializer：使用Jackson 1，将对象序列化为JSON；
- Jackson2JsonRedisSerializer：使用Jackson 2，将对象序列化为JSON；
- JdkSerializationRedisSerializer：使用Java序列化；
- OxmSerializer：使用Spring O/X映射的编排器和解排器（marshaler和unmarshaler）实现序列化，用于XML序列化；
- StringRedisSerializer：序列化String类型的key和value。

这些序列化器都实现了RedisSerializer接口，如果其中没有符合需求的序列化器，那么你还可以自行创建。

RedisTemplate会使用JdkSerializationRedisSerializer，这意味着key和value都会通过Java进行序列化。StringRedisTemplate默认会使用StringRedis-Serializer，这在我们的预料之中，它实际上就是实现String与byte数组之间的相互转换。这些默认的设置适用于很多的场景，但有时候你可能会发现使用一个不同的序列化器也是很有用处的。

例如，假设当使用RedisTemplate的时候，我们希望将Product类型的value序列化为JSON，而key是String类型。RedisTemplate的setKeySerializer()和setValueSerializer()方法就需要如下所示：

```
@Bean
public RedisTemplate<String, Product>
       redisTemplate(RedisConnectionFactory cf) {
  RedisTemplate<String, Product> redis =
      new RedisTemplate<String, Product>();
  redis.setConnectionFactory(cf);
  redis.setKeySerializer(new StringRedisSerializer());
  redis.setValueSerializer(
      new Jackson2JsonRedisSerializer<Product>(Product.class));
  return redis;
}
```

在这里，我们设置RedisTemplate在序列化key的时候，使用StringRedisSerializer，并且也设置了在序列化Product的

12.4 小结

关系型数据库作为数据持久化领域唯一可选方案的时代已经一去不返了。现在，我们有多种不同的数据库，每一种都代表了不同形式的数据，并提供了适应多种领域模型的功能。Spring Data能够让我们在Spring应用中使用这些数据库，并且使用一致的抽象方式访问各种数据库方案。

在本章中，我们基于前一章使用JPA时所学到的Spring Data知识，将其应用到了MongoDB文档数据库和Neo4j图数据库中。与JPA对应的功能类似，Spring Data MongoDB和Spring Data Neo4j项目都提供了基于接口定义自动生成Repository的功能。除此之外，我们还看到了如何使用Spring Data所提供的注解将领域模型映射为文档、节点和关联关系。

Spring Data还支持将数据持久化到Redis key-value存储中。Key-value存储明显要简单一些，因此没有必要支持自动化Repository和映射注解。不过，Spring Data Redis还是提供了两个不同的模板类来使用Redis key-value存储。

不管你选择使用哪种数据库，从数据库中获取数据都是消耗成本的操作。实际上，数据库查询是很多应用最大的性能瓶颈。我们已经看过了如何通过各种数据源存储和获取数据，现在看一下如何避免出现这种瓶颈。在下一章中，我们将会看到如何借助声明式缓存避免不必要的数据库查询。

---

[1] Henry Ford与Samuel Crowther著《我的生活与工作》（Garden City, New York: Garden City Publishing Company, 1922）

第13章　缓存数据

本章内容：

- 启用声明式缓存
- 使用Ehcache、Redis和GemFire实现缓存功能
- 注解驱动的缓存

你有没有遇到过有人反复问你同一个问题的场景，你刚刚给出完解答，马上就会被问相同的问题？我的孩子经常会问我这样的问题：

“我能吃点糖吗？”

“现在几点了？”

“我们到了吗？”

“我能吃点糖吗？”

在很多方面看来，在我们所编写的应用中，有些的组件也是这样的。无状态的组件一般来讲扩展性会更好一些，但它们也会更加倾向于一遍遍地问相同的问题。因为它们是无状态的，所以一旦当前的任务完成，就会丢弃掉已经获取到的所有解答，下一次需要相同的答案时，它们就不得不再问一遍这个问题。

对于所提出的问题，有时候需要一点时间进行获取或计算才能得到答案。我们可能需要在数据库中获取数据，调用远程服务或者执行复杂的计算。为了得到答案，这就会花费时间和资源。

如果问题的答案变更不那么频繁（或者根本不会发生变化），那么按照相同的方式再去获取一遍就是一种浪费了。除此之外，这样做还可能会对应用的性能产生负面的影响。一遍又一遍地问相同的问题，而每次得到的答案都是一样的，与其这样，我们还不如只问一遍并将答案记住，以便稍后再次需要时使用。

缓存（Caching）可以存储经常会用到的信息，这样每次需要的时候，这些信息都是立即可用的。在本章中，我们将会了解到Spring的缓存抽象。尽管Spring自身并没有实现缓存解决方案，但是它对缓存功能提供了声明式的支持，能够与多种流行的缓存实现进行集成。

13.1 启用对缓存的支持

Spring对缓存的支持有两种方式：

- 注解驱动的缓存
- XML声明的缓存

使用Spring的缓存抽象时，最为通用的方式就是在方法上添加`@Cacheable`和`@CacheEvict`注解。在本章中，大多数内容都会使用这种类型的声明式注解。在13.3小节中，我们会看到如何使用XML来声明缓存边界。

在往`bean`上添加缓存注解之前，必须要启用Spring对注解驱动缓存的支持。如果我们使用`Java`配置的话，那么可以在其中的一个配置类上添加`@EnableCaching`，这样的话就能启用注解驱动的缓存。的13.1展现了如何实际使用`@EnableCaching`。

程序清单**13.1**　通过使用**@EnableCaching**启用注解驱动的缓存

```
package com.habuma.cachefun;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.concurrent.ConcurrentMapCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableCaching                                  <— 启用缓存
public class CachingConfig {

  @Bean
  public CacheManager cacheManager() {          <— 声明缓存管理器
    return new ConcurrentMapCacheManager();
  }

}
```

如果以XML的方式配置应用的话，那么可以使用Spring `cache`命名空间中的`<cache:annotation-driven>`元素来启用注解驱动的缓存。

程序清单**13.2**　通过使用启用注解驱动的缓存

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/cache
    http://www.springframework.org/schema/cache/spring-cache.xsd">

  <cache:annotation-driven />                        <—— 启用缓存

  <bean id="cacheManager" class=
  "org.springframework.cache.concurrent.ConcurrentMapCacheManager" />   <—┐
                                                                          │
</beans>                                                    声明缓存管理器 ┘
```

其实在本质上，@EnableCaching和<cache:annotation-driven>的工作方式是相同的。它们都会创建一个切面（aspect）并触发Spring缓存注解的切点（pointcut）。根据所使用的注解以及缓存的状态，这个切面会从缓存中获取数据，将数据添加到缓存之中或者从缓存中移除某个值。

在程序清单13.1和程序清单13.2中，你可能已经注意到了，它们不仅仅启用了注解驱动的缓存，还声明了一个缓存管理器（cache manager）的bean。缓存管理器是Spring缓存抽象的核心，它能够与多个流行的缓存实现进行集成。

在本例中，声明了ConcurrentMapCacheManager，这个简单的缓存管理器使用java.util.concurrent.ConcurrentHashMap作为其缓存存储。它非常简单，因此对于开发、测试或基础的应用来讲，这是一个很不错的选择。但它的缓存存储是基于内存的，所以它的生命周期是与应用关联的，对于生产级别的大型企业级应用程序，这可能并不是理想的选择。

幸好，有多个很棒的缓存管理器方案可供使用。让我们看一下几个最为常用的缓存管理器。

13.1.1 配置缓存管理器

Spring 3.1内置了五个缓存管理器实现，如下所示：

- SimpleCacheManager
- NoOpCacheManager
- ConcurrentMapCacheManager
- CompositeCacheManager
- EhCacheCacheManager

Spring 3.2引入了另外一个缓存管理器，这个管理器可以用在基于JCache（JSR-107）的缓存提供商之中。除了核心的Spring框架，Spring Data又提供了两个缓存管理器：

- `RedisCacheManager`（来自于Spring Data Redis项目）
- `GemfireCacheManager`（来自于Spring Data GemFire项目）

所以可以看到，在为Spring的缓存抽象选择缓存管理器时，我们有很多可选方案。具体选择哪一个要取决于想要使用的底层缓存供应商。每一个方案都可以为应用提供不同风格的缓存，其中有一些会比其他的更加适用于生产环境。尽管所做出的选择会影响到数据如何缓存，但是Spring声明缓存的方式上并没有什么差别。

我们必须选择一个缓存管理器，然后要在Spring应用上下文中，以bean的形式对其进行配置。我们已经看到了如何配置`ConcurrentMapCacheManager`，并且知道它可能并不是实际应用的最佳选择。现在，看一下如何配置Spring其他的缓存管理器，从`EhCacheCacheManager`开始吧。

使用**Ehcache**缓存

Ehcache是最为流行的缓存供应商之一。Ehcache网站上说它是“Java领域应用最为广泛的缓存”。鉴于它的广泛采用，Spring提供集成Ehcache的缓存管理器是很有意义的。这个缓存管理器也就是`EhCacheCacheManager`。

当读这个名字的时候，在cache这个词上似乎有点结结巴巴的感觉。在Spring中配置`EhCacheCacheManager`是很容易的。程序清单13.3展现了如何在Java中对其进行配置。

程序清单**13.3**　以**Java**配置的方式设置**EhCacheCacheManager**

```
package com.habuma.cachefun;
import net.sf.ehcache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.ehcache.EhCacheCacheManager;
import org.springframework.cache.ehcache.EhCacheManagerFactoryBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;

@Configuration
@EnableCaching
public class CachingConfig {
                                                     配置
                                                     EhCacheCacheManager
  @Bean
  public EhCacheCacheManager cacheManager(CacheManager cm) {   <-
     return new EhCacheCacheManager(cm);
  }
                                                     EhCacheManagerFactoryBean
  @Bean
  public EhCacheManagerFactoryBean ehcache() {                 <-
        EhCacheManagerFactoryBean ehCacheFactoryBean =
           new EhCacheManagerFactoryBean();
      ehCacheFactoryBean.setConfigLocation(
           new ClassPathResource("com/habuma/spittr/cache/ehcache.xml"));
      return ehCacheFactoryBean;
    }
```

在程序清单13.3中，cacheManager()方法创建了一个EhCacheCacheManager的实例，这是通过传入Ehcache CacheManager实例实现的。在这里，稍微有点诡异的注入可能会让人感觉迷惑，这是因为Spring和EhCache都定义了CacheManager类型。需要明确的是，EhCache的CacheManager要被注入到Spring的EhCacheCacheManager（Spring CacheManager的实现）之中。

我们需要使用EhCache的CacheManager来进行注入，所以必须也要声明一个CacheManager bean。为了对其进行简化，Spring提供了EhCacheManager-FactoryBean来生成EhCache的CacheManager。方法ehcache()会创建并返回一个EhCacheManagerFactoryBean实例。因为它是一个工厂bean（也就是说，它实现了Spring的FactoryBean接口），所以注册在Spring应用上下文中的并不是EhCacheManagerFactoryBean的实例，而是CacheManager的一个实例，因此适合注入到EhCacheCacheManager之中。

除了在Spring中配置的bean，还需要有针对EhCache的配置。EhCache为XML定义了自己的配置模式，我们需要在一个XML文件中配置缓存，该文件需要符合EhCache所定义的模式。在创建EhCacheManagerFactoryBean的过程中，需要告诉它EhCache配置文件在什么地方。在这里通过调用setConfigLocation()方法，传入ClassPath-Resource，用来指明EhCache XML配置文件

相对于根类路径（classpath）的位置。

至于ehcache.xml文件的内容，不同的应用之间会有所差别，但是至少需要声明一个最小的缓存。例如，如下的EhCache配置声明一个名为spittleCache的缓存，它最大的堆存储为50MB，存活时间为100秒。

```
<ehcache>
  <cache name="spittleCache"
         maxBytesLocalHeap="50m"
         timeToLiveSeconds="100">
  </cache>
</ehcache>
```

显然，这是一个基础的EhCache配置。在你的应用之中，可能需要使用EhCache所提供的丰富的配置选项。参考EhCache的文档以了解调优EhCache配置的细节，地址是http://ehcache.org/documentation/configuration。

使用Redis缓存

如果你仔细想一下的话，缓存的条目不过是一个键值对（key-value pair），其中key描述了产生value的操作和参数。因此，很自然地就会想到，Redis作为key-value存储，非常适合于存储缓存。

Redis可以用来为Spring缓存抽象机制存储缓存条目，Spring Data Redis提供了RedisCacheManager，这是CacheManager的一个实现。RedisCacheManager会与一个Redis服务器协作，并通过RedisTemplate将缓存条目存储到Redis中。

为了使用RedisCacheManager，我们需要RedisTemplate bean以及RedisConnectionFactory实现类（如JedisConnectionFactory）的一个bean。在第12章中，我们已经看到了这些bean该如何配置。在RedisTemplate就绪之后，配置RedisCacheManager就是非常简单的事情了，如程序清单13.4所示。

程序清单**13.4**　配置将缓存条目存储在**Redis**服务器的缓存管理器

```
package com.myapp;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Bean;
import org.springframework.data.redis.cache.RedisCacheManager;
import org.springframework.data.redis.connection.jedis
                                        .JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

@Configuration
@EnableCaching
public class CachingConfig {

  @Bean
  public CacheManager cacheManager(RedisTemplate redisTemplate) {
    return new RedisCacheManager(redisTemplate);          <-- Redis 缓存管理器 bean
   }

  @Bean
  public JedisConnectionFactory redisConnectionFactory() {  <-- Redis 连接工厂 bean
     JedisConnectionFactory jedisConnectionFactory =
            new JedisConnectionFactory();
    jedisConnectionFactory.afterPropertiesSet();
    return jedisConnectionFactory;
  }

  @Bean
  public RedisTemplate<String, String> redisTemplate(     <-- RedisTemplate bean
           RedisConnectionFactory redisCF) {
    RedisTemplate<String, String> redisTemplate =
            new RedisTemplate<String, String>();
    redisTemplate.setConnectionFactory(redisCF);
    redisTemplate.afterPropertiesSet();
    return redisTemplate;
  }

}
```

可以看到，我们构建了一个`RedisCacheManager`，这是通过传递一个`RedisTemplate`实例作为其构造器的参数实现的。

使用多个缓存管理器

我们并不是只能有且仅有一个缓存管理器。如果你很难确定该使用哪个缓存管理器，或者有合法的技术理由使用超过一个缓存管理器的话，那么可以尝试使用Spring的`CompositeCacheManager`。

`CompositeCacheManager`要通过一个或更多的缓存管理器来进行配置，它会迭代这些缓存管理器，以查找之前所缓存的值。以下的程序清单展现了如何创建`CompositeCacheManager` bean，它会迭代`JCacheCacheManager`、`EhCacheCache-Manager`和`RedisCacheManager`。

程序清单**13.5　CompositeCacheManager**会迭代一个缓存管理器的列表

```
@Bean
public CacheManager cacheManager(
      net.sf.ehcache.CacheManager cm,
      javax.cache.CacheManager jcm) {
  CompositeCacheManager cacheManager = new CompositeCacheManager();    <-- 创建 CompositeCacheManager
  List<CacheManager> managers = new ArrayList<CacheManager>();
  managers.add(new JCacheCacheManager(jcm));
  managers.add(new EhCacheCacheManager(cm))
  managers.add(new RedisCacheManager(redisTemplate()));
  cacheManager.setCacheManagers(managers);    <-- 添加单个缓存管理器
  return cacheManager;
}
```

当查找缓存条目时，CompositeCacheManager首先会从JCacheCacheManager开始检查JCache实现，然后通过EhCacheCacheManager检查Ehcache，最后会使用RedisCacheManager来检查Redis，完成缓存条目的查找。

在配置完缓存管理器并启用缓存后，就可以在bean方法上应用缓存规则了。让我们看一下如何使用Spring的缓存注解来定义缓存边界。

13.2 为方法添加注解以支持缓存

如前文所述，Spring的缓存抽象在很大程度上是围绕切面构建的。在Spring中启用缓存时，会创建一个切面，它触发一个或更多的Spring的缓存注解。表13.1列出了Spring所提供的缓存注解。

表13.1中的所有注解都能运用在方法或类上。当将其放在单个方法上时，注解所描述的缓存行为只会运用到这个方法上。如果注解放在类级别的话，那么缓存行为就会应用到这个类的所有方法上。

表13.1 Spring提供了四个注解来声明缓存规则

注　　解	描　　述
@Cacheable	表明Spring在调用方法之前，首先应该在缓存中查找方法的返回值。如果这个值能够找到，就会返回缓存的值。否则的话，这个方法就会被调用，返回值会放到缓存之中
@CachePut	表明Spring应该将方法的返回值放到缓存中。在方法的调用前并不会检查缓存，方法始终都会被调用
@CacheEvict	表明Spring应该在缓存中清除一个或多个条目

@Caching	这是一个分组的注解，能够同时应用多个其他的缓存注解

13.2.1 填充缓存

我们可以看到，@Cacheable和@CachePut注解都可以填充缓存，但是它们的工作方式略有差异。

@Cacheable首先在缓存中查找条目，如果找到了匹配的条目，那么就不会对方法进行调用了。如果没有找到匹配的条目，方法会被调用并且返回值要放到缓存之中。而@CachePut并不会在缓存中检查匹配的值，目标方法总是会被调用，并将返回值添加到缓存之中。

@Cacheable和@CachePut有一些属性是共有的，参见表13.2。

表**13.2** **@Cacheable**和**@CachePut**有一些共有的属性

属　　性	类　型	描　　述
value	String[]	要使用的缓存名称
condition	String	SpEL表达式，如果得到的值是false的话，不会将缓存应用到方法调用上
key	String	SpEL表达式，用来计算自定义的缓存key
unless	String	SpEL表达式，如果得到的值是true的话，返回值不会放到缓存之中

在最简单的情况下，在@Cacheable和@CachePut的这些属性中，只需使用value属性指定一个或多个缓存即可。例如，考虑SpittleRepository的findOne()方法。在初始保存之后，Spittle就不会再发生变化了。如果有的Spittle比较热门并且会被频繁请求，反复地在数据库中进行获取是对时间和资源的浪费。通过在findOne()方法上添加@Cacheable注解，如下面的程序清单所示，能够确保将Spittle保存在缓存中，从而避免对数据

库的不必要访问。

程序清单13.6　通过使用**@Cacheable**，在缓存中存储和获取值

```
@Cacheable("spittleCache")              <— 缓存这个方法的结果
public Spittle findOne(long id) {
  try {
    return jdbcTemplate.queryForObject(
        SELECT_SPITTLE_BY_ID,
        new SpittleRowMapper(),
        id);
  } catch (EmptyResultDataAccessException e) {
    return null;
  }
}
```

当findOne()被调用时，缓存切面会拦截调用并在缓存中查找之前以名spittleCache存储的返回值。缓存的key是传递到findOne()方法中的id参数。如果按照这个key能够找到值的话，就会返回找到的值，方法不会再被调用。如果没有找到值的话，那么就会调用这个方法，并将返回值放到缓存之中，为下一次调用findOne()方法做好准备。

在程序清单13.6中，@Cacheable注解被放到了JdbcSpittleRepository的findOne()方法实现上。这样能够起作用，但是缓存的作用只限于JdbcSpittleRepository这个实现类中，SpittleRepository的其他实现并没有缓存功能，除非也为其添加上@Cacheable注解。因此，可以考虑将注解添加到SpittleRepository的方法声明上，而不是放在实现类中：

```
@Cacheable("spittleCache")
Spittle findOne(long id);
```

当为接口方法添加注解后，@Cacheable注解会被SpittleRepository的所有实现继承，这些实现类都会应用相同的缓存规则。

将值放到缓存之中

@Cacheable会条件性地触发对方法的调用，这取决于缓存中是不是已经有了所需要的值，对于所注解的方法，@CachePut采用了一种更为直接的流程。带有@CachePut注解的方法始终都会被调用，而且它的返回值也会放到缓存中。这提供一种很便利的机制，能够让我们在请求之前预先加载缓存。

例如，当一个全新的Spittle通过SpittleRepository的save()

方法保存之后，很可能马上就会请求这条记录。所以，当save()方法调用后，立即将Spittle塞到缓存之中是很有意义的，这样当其他人通过findOne()对其进行查找时，它就已经准备就绪了。为了实现这一点，可以在save()方法上添加@CachePut注解，如下所示：

```
@CachePut("spittleCache")
Spittle save(Spittle spittle);
```

当save()方法被调用时，它首先会做所有必要的事情来保存Spittle，然后返回的Spittle会被放到spittleCache缓存中。

在这里只有一个问题：缓存的key。如前文所述，默认的缓存key要基于方法的参数来确定。因为save()方法的唯一参数就是Spittle，所以它会用作缓存的key。将Spittle放在缓存中，而它的缓存key恰好是同一个Spittle，这是不是有一点诡异呢？

显然，在这个场景中，默认的缓存key并不是我们想要的。我们需要的缓存key是新保存Spittle的ID，而不是Spittle本身。所以，在这里需要指定一个key而不是使用默认的key。让我们看一下怎样自定义缓存key。

自定义缓存**key**

@Cacheable和@CachePut都有一个名为key属性，这个属性能够替换默认的key，它是通过一个SpEL表达式计算得到的。任意的SpEL表达式都是可行的，但是更常见的场景是所定义的表达式与存储在缓存中的值有关，据此计算得到key。

具体到我们这个场景，我们需要将key设置为所保存Spittle的ID。以参数形式传递给save()的Spittle还没有保存，因此并没有ID。我们只能通过save()返回的Spittle得到id属性。

幸好，在为缓存编写SpEL表达式的时候，Spring暴露了一些很有用的元数据。表13.3列出了SpEL中可用的缓存元数据。

表**13.3　Spring**提供了多个用来定义缓存规则的**SpEL**扩展

表　达　式	描　　述

#root.args	传递给缓存方法的参数，形式为数组
#root.caches	该方法执行时所对应的缓存，形式为数组
#root.target	目标对象
#root.targetClass	目标对象的类，是#root.target.class的简写形式
#root.method	缓存方法
#root.methodName	缓存方法的名字，是#root.method.name的简写形式
#result	方法调用的返回值（不能用在@Cacheable注解上）
#Argument	任意的方法参数名（如#argName）或参数索引（如#a0或#p0）

对于save()方法来说，我们需要的键是所返回Spittle对象的id属性。表达式#result能够得到返回的Spittle。借助这个对象，我们可以通过将key属性设置为#result.id来引用id属性：

```
@CachePut(value="spittleCache", key="#result.id")
Spittle save(Spittle spittle);
```

按照这种方式配置@CachePut，缓存不会去干涉save()方法的执行，但是返回的Spittle将会保存在缓存中，并且缓存的key与Spittle的id属性相同。

条件化缓存

通过为方法添加Spring的缓存注解，Spring就会围绕着这个方法创建一个缓存切面。但是，在有些场景下我们可能希望将缓存功能关闭。

@Cacheable和@CachePut提供了两个属性用以实现条件化缓存：unless和condition，这两个属性都接受一个SpEL表达式。如果unless属性的SpEL表达式计算结果为true，那么缓存方法返回的数据就不会放到缓存中。与之类似，如果condition属性的SpEL表达式计算结果为false，那么对于这个方法缓存就会被禁用

掉。

表面上来看，unless和condition属性做的是相同的事情。但是，这里有一点细微的差别。unless属性只能阻止将对象放进缓存，但是在这个方法调用的时候，依然会去缓存中进行查找，如果找到了匹配的值，就会返回找到的值。与之不同，如果condition的表达式计算结果为false，那么在这个方法调用的过程中，缓存是被禁用的。就是说，不会去缓存进行查找，同时返回值也不会放进缓存中。

作为样例（尽管有些牵强），假设对于message属性包含“NoCache”的Spittle对象，我们不想对其进行缓存。为了阻止这样的Spittle对象被缓存起来，可以这样设置unless属性：

```
@Cacheable(value="spittleCache"
    unless="#result.message.contains('NoCache')")
Spittle findOne(long id);
```

为unless设置的SpEL表达式会检查返回的Spittle对象（在表达式中通过#result来识别）的message属性。如果它包含“NoCache”文本内容，那么这个表达式的计算值为true，这个Spittle对象不会放进缓存中。否则的话，表达式的计算结果为false，无法满足unless的条件，这个Spittle对象会被缓存。

属性unless能够阻止将值写入到缓存中，但是有时候我们希望将缓存全部禁用。也就是说，在一定的条件下，我们既不希望将值添加到缓存中，也不希望从缓存中获取数据。

例如，对于ID值小于10的Spittle对象，我们不希望对其使用缓存。在这种场景下，这些Spittle是用来进行调试的测试条目，对其进行缓存并没有实际的价值。为了要对ID小于10的Spittle关闭缓存，可以在@Cacheable上使用condition属性，如下所示：

```
@Cacheable(value="spittleCache"
    unless="#result.message.contains('NoCache')"
    condition="#id >= 10")
Spittle findOne(long id);
```

如果findOne()调用时，参数值小于10，那么将不会在缓存中进行查找，返回的Spittle也不会放进缓存中，就像这个方法没有添加@Cacheable注解一样。

如样例所示，unless属性的表达式能够通过#result引用返回值。这是很有用的，这么做之所以可行是因为unless属性只有在缓存方法有返回值时才开始发挥作用。而condition肩负着在方法上禁用

缓存的任务，因此它不能等到方法返回时再确定是否该关闭缓存。这意味着它的表达式必须要在进入方法时进行计算，所以我们不能通过#result引用返回值。

我们现在已经在缓存中添加了内容，但是这些内容能被移除掉吗？接下来看一下如何借助@CacheEvict将缓存数据移除掉。

13.2.2 移除缓存条目

@CacheEvict并不会往缓存中添加任何东西。相反，如果带有@CacheEvict注解的方法被调用的话，那么会有一个或更多的条目会在缓存中移除。

那么在什么场景下需要从缓存中移除内容呢？当缓存值不再合法时，我们应该确保将其从缓存中移除，这样的话，后续的缓存命中就不会返回旧的或者已经不存在的值，其中一个这样的场景就是数据被删除掉了。这样的话，SpittleRepository的remove()方法就是使用@CacheEvict的绝佳选择：

```
@CacheEvict("spittleCache")
void remove(long spittleId);
```

> **注意：** 与@Cacheable和@CachePut不同，@CacheEvict能够应用在返回值为void的方法上，而@Cacheable和@CachePut需要非void的返回值，它将会作为放在缓存中的条目。因为@CacheEvict只是将条目从缓存中移除，因此它可以放在任意的方法上，甚至void方法。

从这里可以看到，当remove()调用时，会从缓存中删除一个条目。被删除条目的key与传递进来的spittleId参数的值相等。

@CacheEvict有多个属性，如表13.4所示，这些属性会影响到该注解的行为，使其不同于默认的做法。

可以看到，@CacheEvict的一些属性与@Cacheable和@CachePut是相同的，另外还有几个新的属性。与@Cacheable和@CachePut不同，@CacheEvict并没有提供unless属性。

Spring的缓存注解提供了一种优雅的方式在应用程序的代码中声明缓存规则。但是，Spring还为缓存提供了XML命名空间。在结束对缓存的讨论之前，我们快速地看一下如何以XML的形式配置缓存规则。

表**13.4** **@CacheEvict**注解的属性，指定了哪些缓存条目应该被移除掉

属　性	类型	描　述
value	String[]	要使用的缓存名称
key	String	SpEL表达式，用来计算自定义的缓存key
condition	String	SpEL表达式，如果得到的值是false的话，缓存不会应用到方法调用上
allEntries	boolean	如果为true的话，特定缓存的所有条目都会被移除掉
beforeInvocation	boolean	如果为true的话，在方法调用之前移除条目。如果为false（默认值）的话，在方法成功调用之后再移除条目

13.3　使用XML声明缓存

你可能想要知道为什么想要以XML的方式声明缓存。毕竟，本章中我们所看到的缓存注解要优雅得多。

我认为有两个原因：

- 你可能会觉得在自己的源码中添加Spring的注解有点不太舒服；
- 你需要在没有源码的bean上应用缓存功能。

在上面的任意一种情况下，最好（或者说需要）将缓存配置与缓存数据的代码分隔开来。Spring的cache命名空间提供了使用XML声明缓存规则的方法，可以作为面向注解缓存的替代方案。因为缓存是一种面向切面的行为，所以cache命名空间会与Spring的aop命名空间结合起来使用，用来声明缓存所应用的切点在哪里。

要开始配置XML声明的缓存，首先需要创建Spring配置文件，这个文件中要包含cache和aop命名空间：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/cache
    http://www.springframework.org/schema/cache/spring-cache.xsd">
   <!-- Caching configuration will go here -->

 </beans>
```

cache命名空间定义了在Spring XML配置文件中声明缓存的配置元素。表13.5列出了cache命名空间所提供的所有元素。

表13.5　**Spring**的**cache**命名空间提供了以**XML**方式配置缓存规则的元素

元素	描述
<cache:annotation-driven>	启用注解驱动的缓存。等同于Java配置中的@EnableCaching
<cache:advice>	定义缓存通知（advice）。结合<aop:advisor>，将通知应用到切点上
<cache:caching>	在缓存通知中，定义一组特定的缓存规则
<cache:cacheable>	指明某个方法要进行缓存。等同于@Cacheable注解
<cache:cache-put>	指明某个方法要填充缓存，但不会考虑缓存中是否已有匹配的值。等同于@CachePut注解
<cache:cache-evict>	指明某个方法要从缓存中移除一个或多个条目，等同于@CacheEvict注解

<cache:annotation-driven>元素与Java配置中所对应的@EnableCaching非常类似，会启用注解驱动的缓存。我们已经讨论过这种风格的缓存，因此没有必要再对其进行介绍。

表13.5中其他的元素都用于基于XML的缓存配置。接下来的代码清单

展现了如何使用这些元素为SpittleRepositorybean配置缓存，其作用等同于本章前面章使用缓存注解的方式。

程序清单**13.7**　使用**XML**元素为**SpittleRepository**声明缓存规则

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/cache
    http://www.springframework.org/schema/cache/spring-cache.xsd">

  <aop:config>                                          <-- 将缓存通知绑定
    <aop:advisor advice-ref="cacheAdvice"                   到一个切点上
        pointcut=
          "execution(* com.habuma.spittr.db.SpittleRepository.*(..))"/>
  </aop:config>

  <cache:advice id="cacheAdvice">
    <cache:caching>
      <cache:cacheable                            <-- 配置为支持缓存
          cache="spittleCache"
          method="findRecent" />

      <cache:cacheable                                 <-- 配置为支持缓存
           cache="spittleCache"        method="findOne" />
      <cache:cacheable                           <-- 配置为支持缓存
          cache="spittleCache"
          method="findBySpitterId" />

      <cache:cache-put                           <-- 在 save 时填充缓存
          cache="spittleCache"
          method="save"
          key="#result.id" />

      <cache:cache-evict                         <-- 从缓存中移除
          cache="spittleCache"
          method="remove" />

    </cache:caching>
  </cache:advice>

  <bean id="cacheManager" class=
      "org.springframework.cache.concurrent.ConcurrentMapCacheManager"
  />

</beans>
```

在程序清单13.7中，我们首先看到的是<aop:advisor>，它引用ID为cacheAdvice的通知，该元素将这个通知与一个切点进行匹配，因此建立了一个完整的切面。在本例中，这个切面的切点会在执行SpittleRepository的任意方法时触发。如果这样的方法被Spring应用上下文中的任意某个bean所调用，那么就会调用切面的通知。

在这里，通知利用<cache:advice>元素进行了声明。在<cache:advice>元素中，可以包含任意数量的

<cache:caching>元素，这些元素用来完整地定义应用的缓存规则。在本例中，只包含了一个<cache:caching>元素。这个元素又包含了三个<cache:cacheable>元素和一个<cache:cache-put>元素。

每个<cache:cacheable>元素都声明了切点中的某一个方法是支持缓存的。这是与@Cacheable注解同等作用的XML元素。具体来讲，findRecent()、findOne()和findBySpitterId()都声明为支持缓存，它们的返回值将会保存在名为spittleCache的缓存之中。

<cache:cache-put>是Spring XML中与@CachePut注解同等作用的元素。它表明一个方法的返回值要填充到缓存之中，但是这个方法本身并不会从缓存中获取返回值。在本例中，save()方法用来填充缓存。同面向注解的缓存一样，我们需要将默认的key改为返回Spittle对象的id属性。

最后，<cache:cache-evict>元素是Spring XML中用来替代@CacheEvict注解的。它会从缓存中移除元素，这样的话，下次有人进行查找的时候就找不到了。在这里，调用remove()时，会将缓存中的Spittle删除掉，其中key与remove()方法所传递进来的ID参数相等的条目会从缓存中移除。

需要注意的是，<cache:advice>元素有一个cache-manager元素，用来指定作为缓存管理器的bean。它的默认值是cacheManager，这与程序清单13.7底部的<bean>声明恰好是一致的，所以没有必要再显式地进行设置。但是，如果缓存管理器的ID与之不同的话（使用多个缓存管理器的时候，可能会遇到这样的场景），那么可以通过设置cache-manager属性指定要使用哪个缓存管理器。

另外，还要留意的是，<cache:cacheable>、<cache:cache-put>和<cache:cache-evict>元素都引用了同一个名为spittleCache的缓存。为了消除这种重复，我们可以在<cache:caching>元素上指明缓存的名字：

```
<cache:advice id="cacheAdvice">
  <cache:caching cache="spittleCache">

    <cache:cacheable method="findRecent" />

    <cache:cacheable method="findOne" />

    <cache:cacheable method="findBySpitterId" />

    <cache:cache-put
        method="save"
        key="#result.id" />

    <cache:cache-evict method="remove" />

  </cache:caching>
</cache:advice>
```

<cache:caching>有几个可以供<cache:cacheable>、<cache:cache-put>和<cache:cache-evict>共享的属性，包括：

cache：指明要存储和获取值的缓存；

condition：SpEL表达式，如果计算得到的值为false，将会为这个方法禁用缓存；

key：SpEL表达式，用来得到缓存的key（默认为方法的参数）；

method：要缓存的方法名。

除此之外，<cache:cacheable>和<cache:cache-put>还有一个unless属性，可以为这个可选的属性指定一个SpEL表达式，如果这个表达式的计算结果为true，那么将会阻止将返回值放到缓存之中。

<cache:cache-evict>元素还有几个特有的属性：

- all-entries：如果是true的话，缓存中所有的条目都会被移除掉。如果是false的话，只有匹配key的条目才会被移除掉。
- before-invocation：如果是true的话，缓存条目将会在方法调用之前被移除掉。如果是false的话，方法调用之后才会移除缓存。

all-entries和before-invocation的默认值都是false。这意味着在使用<cache:cache-evict>元素且不配置这两个属性时，会在方法调用完成后只删除一个缓存条目。要删除的条目会通过默认的key（基于方法的参数）进行识别，当然也可以通过为名为key的属性设置一个SpEL表达式指定要删除的key。

13.4 小结

如果想让应用程序避免一遍遍地为同一个问题推导、计算或查询答案的话，缓存是一种很棒的方式。当以一组参数第一次调用某个方法时，返回值会被保存在缓存中，如果这个方法再次以相同的参数进行调用时，这个返回值会从缓存中查询获取。在很多场景中，从缓存查找值会比其他的方式（比如，执行数据库查询）成本更低。因此，缓存会对应用程序的性能带来正面的影响。

在本章中，我们看到了如何在Spring应用中声明缓存。首先，看到的是如何声明一个或更多的Spring缓存管理器。然后，将缓存用到了`Spittr`应用程序中，这是通过将`@Cacheable`、`@CachePut`和`@CacheEvict`添加到`SpittleRepository`上实现的。

我们还看到了如何借助XML将缓存规则的配置与应用程序代码分离开来。`<cache:cacheable>`、`<cache:cache-put>`和`<cache:cache-evict>`元素的作用与本章前面所使用的注解是一致的。

在这个过程中，我们讨论了缓存实际上是一种面向切面的行为。Spring将缓存实现为一个切面。在使用XML声明缓存规则时，这一点非常明显：我们必须要将缓存通知绑定到一个切点上。

Spring在将安全功能应用到方法上时，同样使用了切面。在下一章中，我们将会看到如何借助Spring Security确保bean方法的安全性。

第14章　保护方法应用

本章内容：

- 保护方法调用
- 使用表达式定义安全规则
- 创建安全表达式计算器

在离家或上床睡觉之前，我做的最后一件事就是确保房间的门已经关好。但是在此之前，我会设置好警报。为什么呢？这是因为，尽管门锁是保证安全的一个好办法，但是警报系统提供了第二层防护，窃贼有可能会越过门锁的保护。

在第9章中，我们看到了如何使用Spring Security保护应用的Web层。Web安全是非常重要的，它能阻止用户访问没有权限的内容。但是，如果应用的Web层出现安全漏洞会怎样呢？如果用户能够请求他们不允许访问的内容会怎样呢？

尽管我们没有理由认为用户能够攻破应用的安全层，但是在Web层出现安全漏洞实在是太容易了。例如，假设用户请求了一个允许访问的页面，但是由于开发人员不认真，处理这个请求的控制器方法返回了该用户不允许看到的数据。这是一个无心之失，不过，安全问题很可能就是无心之失所造成的，因为他们是非常聪明的攻击者。

我们可以同时保护应用的Web层以及场景后面的方法，这样就能保证如果用户不具备权限的话，就无法执行相应的逻辑。

在本章中，我们将会看到如何使用Spring Security保护bean方法。通过这种方式，就能声明安全规则，保证如果用户没有执行方法的权限，就不会执行相应的方法。首先，我们会看一些可以放在方法上的简单注解，它们能够将方法锁定，阻止无权限用户的访问。

14.1　使用注解保护方法

在Spring Security中实现方法级安全性的最常见办法是使用特定的注解，将这些注解应用到需要保护的方法上。这样有几个好处，最重要的是当我们在编辑器中查看给定的方法时，能够很清楚地看到它的安全规则。

Spring Security提供了三种不同的安全注解：

- Spring Security自带的@Secured注解；
- JSR-250的@RolesAllowed注解；
- 表达式驱动的注解，包括 @PreAuthorize、@PostAuthorize、@PreFilter和 @PostFilter。

@Secured和@RolesAllowed方案非常类似，能够基于用户所授予的权限限制对方法的访问。当我们需要在方法上定义更灵活的安全规则时，Spring Security提供了@PreAuthorize和@PostAuthorize，而@PreFilter/@PostFilter能够过滤方法返回的以及传入方法的集合。

在本章中，你将会看到如何使用这些注解。作为开始，我们首先看一下@Secured注解，这是Spring Security所提供的方法级安全注解里面最简单的一个。

14.1.1　使用@Secured注解限制方法调用

在Spring中，如果要启用基于注解的方法安全性，关键之处在于要在配置类上使用@EnableGlobalMethodSecurity，如下所示：

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled=true)
public class MethodSecurityConfig
       extends GlobalMethodSecurityConfiguration {
}
```

除了使用@EnableGlobalMethodSecurity注解，我们可能也注意到配置类扩展了GlobalMethodSecurityConfiguration。在第9章中，Web安全的配置类扩展了WebSecurityConfigurerAdapter，与之类似，这个类能够为方法级别的安全性提供更精细的配置。

例如，如果我们在Web层的安全配置中设置认证，那么可以通过重载GlobalMethodSecurityConfiguration的configure()方法实现该功能：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
  auth
  .inMemoryAuthentication()
    .withUser("user").password("password").roles("USER");
}
```

在本章稍后的14.2.2小节中，我们将会看到如何重载GlobalMethodSecurity-Configuration的createExpressionHandler()方法，提供一些自定义的安全表达式处理行为。

让我们回到@EnableGlobalMethodSecurity注解，注意它的securedEnabled属性设置成了true。如果securedEnabled属性的值为true的话，将会创建一个切点，这样的话Spring Security切面就会包装带有@Secured注解的方法。例如，考虑如下这个带有@Secured注解的addSpittle()方法：

```
@Secured("ROLE_SPITTER")
public void addSpittle(Spittle spittle) {
  // ...
}
```

@Secured注解会使用一个String数组作为参数。每个String值是一个权限，调用这个方法至少需要具备其中的一个权限。通过传递进来ROLE_SPITTER，我们告诉Spring Security只允许具有ROLE_SPITTER权限的认证用户才能调用addSpittle ()方法。

如果传递给@Secured多个权限值，认证用户必须至少具备其中的一个才能进行方法的调用。例如，下面使用@Secured的方式表明用户必须具备ROLE_SPITTER或ROLE_ADMIN权限才能触发这个方法：

```
@Secured({"ROLE_SPITTER", "ROLE_ADMIN"})
public void addSpittle(Spittle spittle) {
  // ...
}
```

如果方法被没有认证的用户或没有所需权限的用户调用，保护这个方法的切面将抛出一个Spring Security异常（可能是AuthenticationException或AccessDeniedException的子类）。它们是非检查型异常，但这个异常最终必须要被捕获和处理。如果被保护的方法是在Web请求中调用的，这个异常会被Spring Security的过滤器自动处理。否则的话，你需要编写代码来处理这个异常。

@Secured注解的不足之处在于它是Spring特定的注解。如果更倾向于使用Java标准定义的注解，那么你应该考虑使用@RolesAllowed注解。

14.1.2 在Spring Security中使用JSR-250的@RolesAllowed注解

@RolesAllowed注解和@Secured注解在各个方面基本上都是一致的。唯一显著的区别在于@RolesAllowed是JSR-250定义的Java标准注解。

差异更多在于政治考量而非技术因素。但是，当使用其他框架或API来处理注解的话，使用标准的@RolesAllowed注解会更有意义。

如果选择使用@RolesAllowed的话，需要将@EnableGlobalMethodSecurity的jsr250Enabled属性设置为true，以开启此功能：

```
@Configuration
@EnableGlobalMethodSecurity(jsr250Enabled=true)
public class MethodSecurityConfig
       extends GlobalMethodSecurityConfiguration {
}
```

尽管我们这里只是启用了jsr250Enabled，但需要说明的一点是这与securedEnabled并不冲突。这两种注解风格可以同时启用。

在将jsr250Enabled设置为true之后，将会启用一个切点，这样带有@RolesAllowed注解的方法都会被Spring Security的切面包装起来。因此，在方法上使用@RolesAllowed的方式与使用@Secured类似。例如，如下的addSpittle()方法使用了@RolesAllowed注解来代替@Secured：

```
@RolesAllowed("ROLE_SPITTER")
public void addSpittle(Spittle spittle) {
  // ...
}
```

尽管@RolesAllowed比 @Secured在政治上稍微有点优势，它是实现方法安全的标准注解，但是这两个注解有一个共同的不足。它们只能根据用户有没有授予特定的权限来限制方法的调用。在判断方式是否执行方面，无法使用其他的因素。我们在第9章曾经看到过，在保护URL方面，能够使用SpEL表达式克服这一限制。接下来，我们看一下如何组合使用SpEL与Spring Security所提供的方法调用前后注解，

实现基于表达式的方法安全性。

14.2 使用表达式实现方法级别的安全性

尽管@Secured和@RolesAllowed注解在拒绝未认证用户方面表现不错，但这也是它们所能做到的所有事情了。有时候，安全性约束不仅仅涉及用户是否有权限。

Spring Security 3.0引入了几个新注解，它们使用SpEL能够在方法调用上实现更有意思的安全性约束。这些新的注解在表14.1中进行了描述。

这些注解的值参数中都可以接受一个SpEL表达式。表达式可以是任意合法的SpEL表达式，可能会包含表9.5所列的Spring Security对SpEL的扩展。如果表达式的计算结果为true，那么安全规则通过，否则就会失败。安全规则通过或失败的结果会因为所使用注解的差异而有所不同。

表**14.1** **Spring Security 3.0**提供了**4**个新的注解，可以使用**SpEL**表达式来保护方法调用

注　　解	描　　述
@PreAuthorize	在方法调用之前，基于表达式的计算结果来限制对方法的访问
@PostAuthorize	允许方法调用，但是如果表达式计算结果为false，将抛出一个安全性异常
@PostFilter	允许方法调用，但必须按照表达式来过滤方法的结果
@PreFilter	允许方法调用，但必须在进入方法之前过滤输入值

稍后，我们将会看到每个注解的例子。但首先，我们需要将@EnableGlobalMethod-Security注解的prePostEnabled属性设置为true，从而启用它们：

```
@Configuration
public class MethodSecurityConfig
       extends GlobalMethodSecurityConfiguration {
}
```

现在，方法调用前后的注解都已经启用了，我们可以使用它们了。我们首先看一下如何使用@PreAuthorize和@PostAuthorize注解限制对方法的调用。

14.2.1 表述方法访问规则

到目前为止，我们已经看到@Secured和@RolesAllowed能够限制只有用户具备所需的权限才能触发方法的执行。但是，这两个注解的不足在于它们只能基于用户授予的权限来做出决策。

Spring Security还提供了两个注解，@PreAuthorize和@PostAuthorize，它们能够基于表达式的计算结果来限制方法的访问。在定义安全限制方面，表达式带了极大的灵活性。通过使用表达式，只要我们能够想象得到，就可以定义任意允许访问或不允许访问方法的条件。

@PreAuthorize和@PostAuthorize之间的关键区别在于表达式执行的时机。@PreAuthorize的表达式会在方法调用之前执行，如果表达式的计算结果不为true的话，将会阻止方法执行。与之相反，@PostAuthorize的表达式直到方法返回才会执行，然后决定是否抛出安全性的异常。

在方法调用前验证权限

@PreAuthorize乍看起来可能只是添加了SpEL支持的@Secured和@RolesAllowed。实际上，你可以基于用户所授予的角色，使用@PreAuthorize来限制访问：

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
public void addSpittle(Spittle spittle) {
  // ...
}
```

如果按照这种方式的话，@PreAuthorize相对于@Secured和@RolesAllowed并没有什么优势。如果用户具有ROLE_SPITTER角色的话，允许方法调用。否则，将会抛出安全性异常，方法也不会执行。

但是，@PreAuthorize的功能并不限于这个简单例子所展现

的。@PreAuthorize的String类型参数是一个SpEL表达式。借助于SpEL表达式来实现访问决策，我们能够编写出更高级的安全性约束。例如，Spittr应用程序的一般用户只能写140个字以内的Spittle，而付费用户不限制字数。

虽然@Secured和@RolesAllowed在这里无能为力，但是@PreAuthorize注解恰好能够适用于这种场景：

```
@PreAuthorize(
     "(hasRole('ROLE_SPITTER') and #spittle.text.length() <= 140)"
     +"or hasRole('ROLE_PREMIUM')")
public void addSpittle(Spittle spittle) {
  // ...
}
```

表达式中的#spittle部分直接引用了方法中的同名参数。这使得Spring Security能够检查传入方法的参数，并将这些参数用于认证决策的制定。在这里，我们深入到Spitter的文本内容中，保证不超过Spittr标准用户的长度限制。如果是付费用户，那么就没有长度限制了。

在方法调用之后验证权限

在方法调用之后验证权限并不是比较常见的方式。事后验证一般需要基于安全保护方法的返回值来进行安全性决策。这种情况意味着方法必须被调用执行并且得到了返回值。

例如，假设我们想对getSpittleById()方法进行保护，确保返回的Spittle对象属于当前的认证用户。我们只有得到Spittle对象之后，才能判断它是否属于当前用户。因此，getSpittleById()方法必须要先执行。在得到Spittle之后，如果它不属于当前用户的话，将会抛出安全性异常。

除了验证的时机之外，@PostAuthorize与@PreAuthorize的工作方式差不多，只不过它会在方法执行之后，才会应用安全规则。此时，它才有机会在做出安全决策时，考虑到返回值的因素。

例如，要保护上面描述的getSpittleById()方法，我们可以按照如下的方式使用@PostAuthorize注解：

```
@PostAuthorize("returnObject.spitter.username == principal.username")
public Spittle getSpittleById(long id) {
  // ...
}
```

为了便利地访问受保护方法的返回对象，Spring Security在SpEL中提供了名为returnObject的变量。在这里，我们知道返回对象是一个Spittle对象，所以这个表达式可以直接访问其spittle属性中的username属性。

在对比表达式双等号的另一侧，表达式到内置的principal对象中取出其username属性。principal是另一个Spring Security内置的特殊名称，它代表了当前认证用户的主要信息（通常是用户名）。

在Spittle对象所包含Spitter中，如果username属性与principal的username属性相同，这个Spittle将返回给调用者。否则，会抛出一个AccessDeniedException异常，而调用者也不会得到Spittle对象。

有一点需要注意，不像@PreAuthorize注解所标注的方法那样，@PostAuthorize注解的方法会首先执行然后被拦截。这意味着，你需要小心以保证如果验证失败的话不会有一些负面的结果。

14.2.2 过滤方法的输入和输出

如果我们希望使用表达式来保护方法的话，那使用@PreAuthorize和@PostAuthorize是非常好的方案。但是，有时候限制方法调用太严格了。有时，需要保护的并不是对方法的调用，需要保护的是传入方法的数据和方法返回的数据。

例如，我们有一个名为getOffensiveSpittles()的方法，这个方法会返回标记为具有攻击性的Spittle列表。这个方法主要会给管理员使用，以保证Spittr应用中内容的和谐。但是，普通用户也可以使用这个方法，用来查看他们所发布的Spittle有没有被标记为具有攻击性。这个方法的签名大致如下所示；

```
public List<Spittle> getOffensiveSpittles() { ... }
```

按照这种方法的定义，getOffensiveSpittles()方法与具体的用户并没有关联。它只会返回攻击性Spittle的一个列表，并不关心它们属于哪个用户。对于管理员使用来说，这是一个很好的方法，但是它无法限制列表中的Spittle都属于当前用户。

当然，我们也可以重载getOffensiveSpittles()，实现另一个版本，让它接受一个用户ID作为参数，查询给定用户的Spittle。但是，正如我在本章开头所讲的那样，始终会有这样的可能性，那就

是将较为宽松限制的版本用在具有一定安全限制的场景中。[1]

我们需要有一种方式过滤getOffensiveSpittles()方法返回的Spittle集合，将结果限制为允许当前用户看到的内容，而这就是Spring Security的@PostFilter所能做的事情。我们来试一下。

事后对方法的返回值进行过滤

与@PreAuthorize和@PostAuthorize类似，@PostFilter也使用一个SpEL作为值参数。但是，这个表达式不是用来限制方法访问的，@PostFilter会使用这个表达式计算该方法所返回集合的每个成员，将计算结果为false的成员移除掉。

为了阐述该功能，我们将@PostFilter应用在getOffensiveSpittles()方法上：

```
@PreAuthorize("hasAnyRole({'ROLE_SPITTER', 'ROLE_ADMIN'})")
@PostFilter( "hasRole('ROLE_ADMIN') || "
          + "filterObject.spitter.username == principal.name")
public List<Spittle> getOffensiveSpittles() {
  ...
}
```

在这里，@PreAuthorize限制只有具备ROLE_SPITTER或ROLE_ADMIN权限的用户才能访问该方法。如果用户能够通过这个检查点，那么方法将会执行，并且会返回Spittle所组成的一个List。但是，@PostFilter注解将会过滤这个列表，确保用户只能看到允许的Spittle。具体来讲，管理员能够看到所有攻击性的Spittle，非管理员只能看到属于自己的Spittle。

表达式中的filterObject对象引用的是这个方法所返回List中的某一个元素（我们知道它是一个Spittle）。在这个Spittle对象中，如果Spitter的用户名与认证用户（表达式中的principal.name）相同或者用户具有ROLE_ADMIN角色，那这个元素将会最终包含在过滤后的列表中。否则，它将被过滤掉。

事先对方法的参数进行过滤

除了事后过滤方法的返回值，我们还可以预先过滤传入到方法中的值。这项技术不太常用，但是在有些场景下可能会很便利。

例如，假设我们希望以批处理的方式删除Spittle组成的列表。为了完成该功能，我们可能会编写一个方法，其签名大致如下所示：

```
public void deleteSpittles(List<Spittle> spittles) { ... }
```

看起来很简单，对吧？但是，如果我们想在它上面应用一些安全规则的话，比如Spittle只能由其所有者或管理员删除，那该怎么做呢？如果是这样的话，我们可以将逻辑放在deleteSpittles()方法中，在这里循环列表中的Spittle，只删除属于当前用户的那一部分对象（如果当前用户是管理员的话，则会全部删除）。

这能够运行正常，但是这意味着我们需要将安全逻辑直接嵌入到方法之中。相对于删除Spittle来讲，安全逻辑是独立的关注点（当然，它们也有所关联）。如果列表中能够只包含实际要删除的Spittle，这样会更好一些，因为这能帮助deleteSpittles()方法中的逻辑更加简单，只关注于删除Spittle的任务。

Spring Security的@PreFilter注解能够很好地解决这个问题。与@PostFilter非常类似，@PreFilter也使用SpEL来过滤集合，只有满足SpEL表达式的元素才会留在集合中。但是它所过滤的不是方法的返回值，@PreFilter过滤的是要进入方法中的集合成员。

@PreFilter的使用非常简单。如下的deleteSpittles()方法使用了@PreFilter注解：

```
@PreAuthorize("hasAnyRole({'ROLE_SPITTER', 'ROLE_ADMIN'})")
@PreFilter( "hasRole('ROLE_ADMIN') || "
        + "targetObject.spitter.username == principal.name")
public void deleteSpittles(List<Spittle> spittles) { ... }
```

与前面一样，对于没有ROLE_SPITTER或ROLE_ADMIN权限的用户，@PreAuthorize注解会阻止对这个方法的调用。但同时，@PreFilter注解能够保证传递给deleteSpittles()方法的列表中，只包含当前用户有权限删除的Spittle。这个表达式会针对集合中的每个元素进行计算，只有表达式计算结果为true的元素才会保留在列表中。targetObject是Spring Security提供的另外一个值，它代表了要进行计算的当前列表元素。

Spring Security提供了注解驱动的功能，这是通过一系列注解来实现的，到此为止，我们已经对这些注解进行了介绍。相对于判断用户所授予的权限，使用表达式来定义安全限制是一种更为强大的方式。

即便如此，我们也不应该让表达式过于聪明智能。我们应该避免编写非常复杂的安全表达式，或者在表达式中嵌入太多与安全无关的业务逻辑。而且，表达式最终只是一个设置给注解的String值，因此它

很难测试和调试。

如果你觉得自己的安全表达式难以控制了，那么就应该看一下如何编写自定义的许可计算器（permission evaluator），以简化你的SpEL表达式。下面我们看一下如何编写自定义的许可计算器，用它来简化之前用于过滤的表达式。

定义许可计算器

我们在@PreFilter和@PostFilter中所使用的表达式还算不上太复杂。但是，它也并不简单，我们可以很容易地想象如果还要实现其他的安全规则，这个表达式会不断膨胀。在变得很长之前，表达式就会笨重、复杂且难以测试。

其实我们能够将整个表达式替换为更加简单的版本，如下所示：

```
@PreAuthorize("hasAnyRole({'ROLE_SPITTER', 'ROLE_ADMIN'})")
@PreFilter("hasPermission(targetObject, 'delete')")
public void deleteSpittles(List<Spittle> spittles) { ... }
```

现在，设置给@PreFilter的表达式更加紧凑。它实际上只是在问一个问题“用户有权限删除目标对象吗？”。如果有的话，表达式的计算结果为true，Spittle会保存在列表中，并传递给deleteSpittles()方法。如果没有权限的话，它将会被移除掉。

但是，hasPermission()是哪来的呢？它的意思是什么？更为重要的是，它如何知道用户有没有权限删除targetObject所对应的Spittle呢？

hasPermission()函数是Spring Security为SpEL提供的扩展，它为开发者提供了一个时机，能够在执行计算的时候插入任意的逻辑。我们所需要做的就是编写并注册一个自定义的许可计算器。程序清单14.1展现了SpittlePermissionEvaluator类，它就是一个自定义的许可计算器，包含了表达式逻辑。

程序清单**14.1**　许可计算器为**hasPermission()**提供实现逻辑

```
package spittr.security;
import java.io.Serializable;
import org.springframework.security.access.PermissionEvaluator;
import org.springframework.security.core.Authentication;
import spittr.Spittle;

public class SpittlePermissionEvaluator implements PermissionEvaluator {
  private static final GrantedAuthority ADMIN_AUTHORITY =
      new GrantedAuthorityImpl("ROLE_ADMIN");

  public boolean hasPermission(Authentication authentication,
          Object target, Object permission) {

      if (target instanceof Spittle) {
          Spittle spittle = (Spittle) target;
          String username = spittle.getSpitter().getUsername();
          if ("delete".equals(permission)) {
              return isAdmin(authentication) ||
                username.equals(authentication.getName());
          }
      }

      throw new UnsupportedOperationException(
              "hasPermission not supported for object <" + target
                      + "> and permission <" + permission + ">");
  }

  public boolean hasPermission(Authentication authentication,
      Serializable targetId, String targetType, Object permission) {
      throw new UnsupportedOperationException();
  }

  private boolean isAdmin(Authentication authentication) {
      return authentication.getAuthorities().contains(ADMIN_AUTHORITY);
  }

}
```

SpittlePermissionEvaluator实现了Spring Security的PermissionEvaluator接口，它需要实现两个不同的hasPermission()方法。其中的一个hasPermission()方法把要评估的对象作为第二个参数。第二个hasPermission()方法在只有目标对象的ID可以得到的时候才有用，并将ID作为Serializable传入第二个参数。

为了满足我们的需求，我们假设使用Spittle对象来评估权限，所以第二个方法只是简单地抛出UnsupportedOperationException。

对于第一个hasPermission()方法，要检查所评估的对象是否为一个Spittle，并判断所检查的是否为删除权限。如果是这样，它将对比Spitter的用户名是否与认证用户的名称相等，或者当前用户是否具有ROLE_ADMIN权限。

许可计算器已经准备就绪，接下来需要将其注册到Spring Security中，以便在使用@PreFilter表达式的时候支持hasPermission()操

作。为了实现该功能，我们需要替换原有的表达式处理器，换成使用自定义许可计算器的处理器。

默认情况下，Spring Security会配置为使用DefaultMethodSecurityExpression-Handler，它会使用一个DenyAllPermissionEvaluator实例。顾名思义，Deny-AllPermissionEvaluator将会在hasPermission()方法中始终返回false，拒绝所有的方法访问。但是，我们可以为Spring Security提供另外一个DefaultMethod-SecurityExpressionHandler，让它使用我们自定义的SpittlePermissionEvaluator，这需要重载GlobalMethodSecurityConfiguration的createExpressionHandler方法：

```
@Override
protected MethodSecurityExpressionHandler createExpressionHandler() {
  DefaultMethodSecurityExpressionHandler expressionHandler =
      new DefaultMethodSecurityExpressionHandler();
  expressionHandler.setPermissionEvaluator(
      new SpittlePermissionEvaluator());
  return expressionHandler;
}
```

现在，我们不管在任何地方的表达式中使用hasPermission()来保护方法，都会调用SpittlePermissionEvaluator来决定用户是否有权限调用方法。

14.3 小结

方法级别的安全性是Spring Security Web级别安全性的一个重要补充，我们曾在第9章讨论过Web安全性。对于非Web应用来说，方法级别的安全性则是最前沿的防护。对于Web应用来讲，基于安全规则所声明的方法级别安全性能够保护Web请求。

在本章中，我们看到了六个可以在方法上声明安全性限制的注解。对于简单场景来说，面向权限的注解，包括Spring Security的@Secured以及基于标准的@RolesAllowed都很便利。当安全规则更为复杂的时候，组合使用@PreAuthorize、@PostAuthorize以及SpEL能够发挥更强大的威力。我们还看到通过为@PreFilter和@PostFilter提供SpEL表达式，过滤方法的输入和输出。

最后，我们还看到了让安全规则更加易于维护、测试和调试的方法，那就是自定义表达式计算器，它能够用在SpEL表达式的

`hasPermission()`函数中。

从下一章开始，我们将会转移方向，从使用Spring开发后端应用程序转向与其他应用集成。在接下来的几章中，我们将会看到各种集成技术，包括远程服务、异步消息、REST甚至还有发送E-mail。在下一章我们将会探讨第一项集成技术，也就是使用Spring远程服务。

---

[1]除此之外，如果重载getOffensiveSpittles()方法的话，我必须再绞尽脑汁想一个例子出来，以展现如何使用SpEL过滤方法的输出。

第4部分　Spring集成

应用程序都不是孤岛。如今，企业级应用程序必须要与其他的系统协作才能完成其目标。在第4部分，你将会学到如何跨越应用程序本身的边界，与其他的应用程序和企业级服务实现集成。

在第15章“使用远程服务”中，你会学到如何将应用程序中的对象导出为远程服务，还会学习如何透明地访问远程服务，这些服务就像是应用程序中的其他对象一样。我们将会介绍各种远程技术，包括RMI、Hessian/Burlap以及使用JAX-WS的SOAP Web服务。

与第15章所介绍的RPC风格的远程服务不同，第16章“使用Spring MVC创建REST API”将会探讨如何使用Spring MVC构建RESTful服务，它关注于应用程序中的资源。

第17章“Spring消息”将会探索一种不同的应用集成方式，也就是Spring如何用于Java消息服务（Java Message Service，JMS）和高级消息队列协议（Advanced Message Queuing Protocol，AMQP），从而实现应用程序之间的异步通信。

Web应用需要越来越多的交互性，我们希望它能展现实时的数据。第18章“使用WebSocket和STOMP实现消息功能”将会展现Spring的一项新功能，它支持在服务器和Web客户端之间实现异步通信。

另外一种形式的异步通信不一定发生在应用程序之间。在第19章“使用Spring发送Email”中，将会展现如何借助Spring以Email的形式发送异步消息给目标人群。

管理和监控Spring bean是第20章“使用JMX管理Spring Bean”的主题。在该章中，你会学到如何把配置在Spring中bean自动导出为JMX MBean。

本书的结尾是很新但是很必要的内容。第21章“借助Spring Boot简化Spring开发”介绍了在Spring开发中一个令人兴奋且能够改变游戏规则的项目。在典型的Spring应用中，会有很多繁杂的样板式配置，在这一章将会看到Spring Boot如何移除这些配置，能够让我们关注于业务功能的实现。

第15章　使用远程服务

本章内容：

- 访问和发布RMI服务
- 使用Hessian和Burlap服务
- 使用Spring的HTTP invoker
- 使用Spring开发Web服务

想象一下，我们被困在一个荒凉的小岛上，这听上去就像是一场梦境变成了现实。毕竟，谁不想在海滩上静静地独处，可以幸福地不顾外面世界的纷纷扰扰呢？

但是在一个荒岛上，我们不可能总是享受冰镇果汁朗姆酒和日光浴，就算我们能享受这样宁静的隐居生活，但是过不了多久我们就会感到饥饿、厌烦和孤独。在这样的时光里，我们只能以椰子和用叉子所捕的鱼为生。我们终究还是需要食物、新的衣服以及其他供给。而且如果不能和其他人取得联系，不久我们就只能和排球说话了！

我们开发的很多应用就像被遗弃的荒岛。表面上看，它们好像能自给自足，但实际上，它们可能还需要和其他系统相互合作，这些系统既包括组织内部的也包括组织外部的。

例如，采购系统需要与厂商的供应链系统通信；公司的人力资源系统可能需要集成薪金系统；或者，薪金系统需要和打印、邮寄工资等外部系统进行通信。无论哪种情况，我们的应用都需要和其他系统进行交互，远程访问它们的服务。

作为一个Java开发者，我们有多种可以使用的远程调用技术，包括：

- 远程方法调用（Remote Method Invocation，RMI）；
- Caucho的Hessian和Burlap；
- Spring基于HTTP的远程服务；
- 使用JAX-RPC和JAX-WS的Web Service。

不管我们选择哪种远程调用技术，Spring为使用这几种不同的技术访问和创建远程服务都提供了广泛的支持。在本章，我们将学习Spring如何简化和完善这些远程调用服务。但是首先，让我们先简要了解一下远程调用是如何在Spring中工作的。

15.1 Spring远程调用概览

远程调用是客户端应用和服务端之间的会话。在客户端，它所需要的一些功能并不在该应用的实现范围之内，所以应用要向能提供这些功能的其他系统寻求帮助。而远程应用通过远程服务暴露这些功能。

假设我们想把Spittr应用中的某些功能发布为远程服务并提供给其他应用来使用。或许除了现有的基于浏览器的用户界面，我们还想为Spittr应用提供桌面应用或移动端应用，如图15.1所示。为了实现此想法，我们需要把`SpitterService`接口的基本功能发布为远程服务。

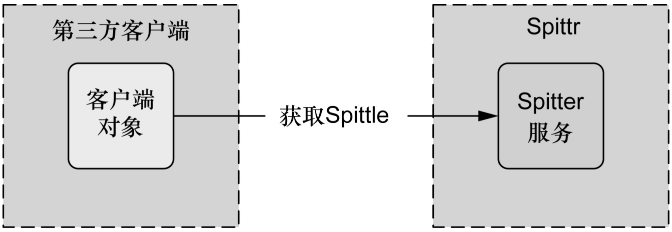

图15.1　第三方客户端能够远程调用Spittr的服务，从而实现与Spittr应用交互

其他应用与Spittr之间的会话开始于客户端应用的一个远程过程调用（remote procedure call，RPC）。从表面上看，RPC类似于调用一个本地对象的一个方法。这两者都是同步操作，会阻塞调用代码的执行，直到被调用的过程执行完毕。

它们的差别仅仅是距离的问题，类似于人与人之间的交流。如果我们在公共场所的饮水机旁讨论周末足球比赛的结果，那我们就是在进行一个本地会话——两人之间的会话发生在同一房间内。同样，本地方法调用是指同一个应用中的两个代码块之间的执行流交换。

另一方面，如果我们拿起电话打给另一个城市的客户端，那我们之间的会话就是通过电话网络远程进行的。类似地，RPC调用就是执行流从一个应用传递给另一个应用，理论上另一个应用部署在跨网络的一台远程机器上。

正如我之前所述，Spring支持多种不同的RPC模型，包括RMI、Caucho的Hessian和Burlap以及Spring自带的HTTP invoker。表15.1概述了每一个RPC模型，并简要讨论了它们所适用的不同场景。

表**15.1**　**Spring**通过多种远程调用技术支持**RPC**

RPC模型	适用场景
远程方法调用（RMI）	不考虑网络限制时（例如防火墙），访问/发布基于Java的服务
Hessian或Burlap	考虑网络限制时，通过HTTP访问/发布基于Java的服务。Hessian是二进制协议，而Burlap是基于XML的
HTTP invoker	考虑网络限制，并希望使用基于XML或专有的序列化机制实现Java序列化时，访问/发布基于Spring的服务
JAX-RPC和JAX-WS	访问/发布平台独立的、基于SOAP的Web服务

不管你选择哪种远程调用模型，我们会发现Spring都提供了风格一致的支持。这意味着一旦理解了如何配置Spring来使用其中的一种模型，如果我们决定使用另外一种模型的话，将拥有非常低的学习曲线。

在所有的模型中，服务都作为Spring所管理的bean配置到我们的应用中。这是通过一个代理工厂bean实现的，这个bean能够把远程服务像本地对象一样装配到其他bean的属性中去。图15.2展示了它是如何工作的。

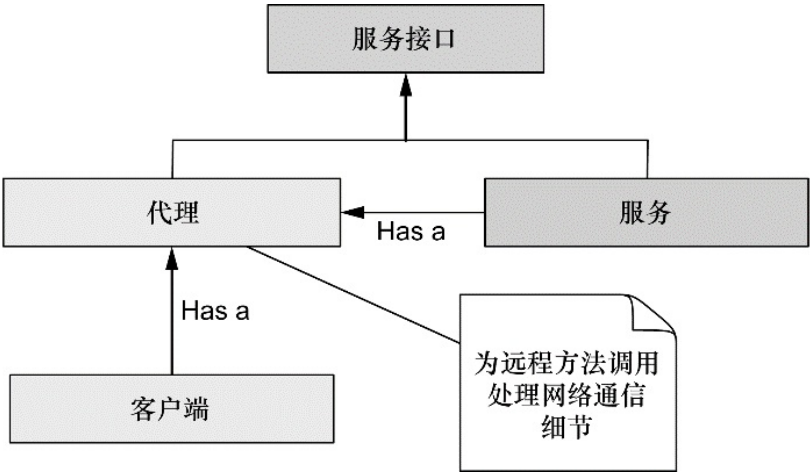

图15.2　在Spring中，远程服务被代理，所以它们能够像其他Spring bean一样被装配到客户端代码中

客户端向代理发起调用，就像代理提供了这些服务一样。代理代表客

户端与远程服务进行通信，由它负责处理连接的细节并向远程服务发起调用。

更重要的是，如果调用远程服务时发生java.rmi.RemoteException异常，代理会处理此异常并重新抛出非检查型异常RemoteAccessException。远程异常通常预示着系统发生了无法优雅恢复的问题，如网络或配置问题。既然客户端通常无法从远程异常中恢复，那么重新抛出RemoteAccessException异常就能让客户端来决定是否处理此异常。

在服务器端，我们可以使用表15.1所列出的任意一种模型将Spring管理的bean发布为远程服务。图15.3展示了远程导出器（remote exporter）如何将bean方法发布为远程服务。

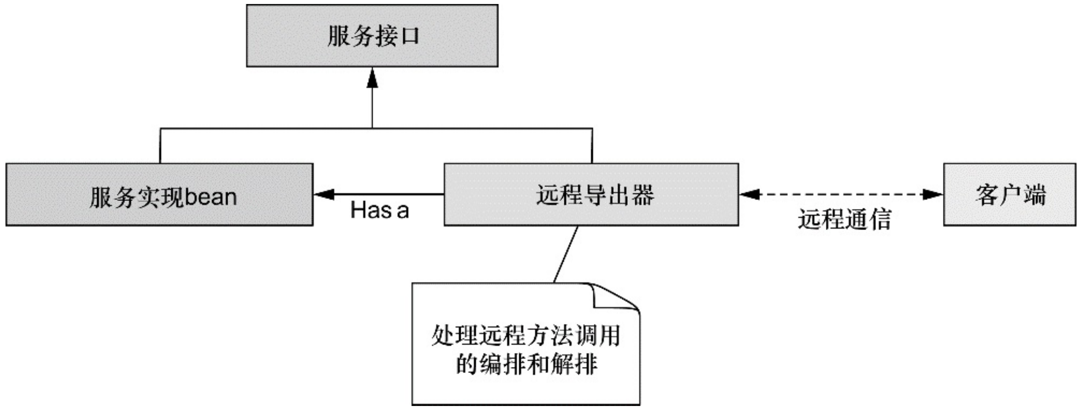

图15.3　使用远程导出器将Spring管理的bean发布为远程服务

无论我们开发的是使用远程服务的代码，还是实现这些服务的代码，或者两者兼而有之，在Spring中，使用远程服务纯粹是一个配置问题。我们不需要编写任何Java代码就可以支持远程调用。我们的服务bean也不需要关心它们是否参与了一个RPC（当然，任何传递给远程调用的bean或从远程调用返回的bean可能需要实现java.io.Serializable接口）。

让我们通过RMI——Java最初的远程调用技术——来开始探索Spring对远程调用的支持吧。

15.2　使用RMI

如果你已经使用Java编程有些年头的话，你肯定会听说过（也可能使

用过）RMI。RMI最初在JDK 1.1被引入到Java平台中，它为Java开发者提供了一种强大的方法来实现Java程序间的交互。在RMI之前，对于Java开发者来说，远程调用的唯一选择就是CORBA（在当时，需要购买一种第三方产品，叫作*Object Request Broker*[ORB]），或者手工编写Socket程序。

但是开发和访问RMI服务是非常乏味无聊的，它涉及到好几个步骤，包括程序的和手工的。Spring简化了RMI模型，它提供了一个代理工厂bean，能让我们把RMI服务像本地JavaBean那样装配到我们的Spring应用中。Spring还提供了一个远程导出器，用来简化把Spring管理的bean转换为RMI服务的工作。

对于Spittr应用，我们将展示如何把一个RMI服务装配进客户端应用程序的Spring应用上下文中。但首先，让我们看看如何使用RMI导出器把`SpitterService`的实现发布为RMI服务。

15.2.1 导出RMI服务

如果你曾经创建过RMI服务，应该会知道这会涉及如下几个步骤：

1．编写一个服务实现类，类中的方法必须抛出`java.rmi.RemoteException`异常；

2．创建一个继承于`java.rmi.Remote`的服务接口；

3．运行RMI编译器（`rmic`），创建客户端stub类和服务端skeleton类；

4．启动一个RMI注册表，以便持有这些服务；

5．在RMI注册表中注册服务。

哇！发布一个简单的RMI服务需要做这么多的工作。除了这些必需的步骤外，你可能注意到了，会抛出相当多的`RemoteException`和`MalformedURLException`异常。虽然这些异常通常意味着一个无法从`catch`代码块中恢复的致命错误，但是我们仍然需要编写样板式的代码来捕获并处理这些异常——即使我们不能修复它们。

很明显，发布一个RMI服务涉及到大量的代码和手工作业。Spring是否能够做一些工作来让这些事情变得不再那么棘手呢？

在Spring中配置RMI服务

幸运的是，Spring提供了更简单的方式来发布RMI服务，不用再编写那些需要抛出RemoteException异常的特定RMI类，只需简单地编写实现服务功能的POJO就可以了，Spring会处理剩余的其他事项。

我们将要创建的RMI服务需要发布SpitterService接口中的方法，如下的程序清单展现了该接口定义。

程序清单**15.1　SpitterService**定义了**Spittr**应用的服务层

```
package com.habuma.spittr.service;
import java.util.List;
import com.habuma.spittr.domain.Spitter;
import com.habuma.spittr.domain.Spittle;
public interface SpitterService {
  List<Spittle> getRecentSpittles(int count);
  void saveSpittle(Spittle spittle);
  void saveSpitter(Spitter spitter);
  Spitter getSpitter(long id);
  void startFollowing(Spitter follower, Spitter followee);
  List<Spittle> getSpittlesForSpitter(Spitter spitter);
  List<Spittle> getSpittlesForSpitter(String username);
  Spitter getSpitter(String username);
  Spittle getSpittleById(long id);
  void deleteSpittle(long id);
  List<Spitter> getAllSpitters();
}
```

如果我们使用传统的RMI来发布此服务，SpitterService和SpitterServiceImpl中的所有方法都需要抛出java.rmi.RemoteException。但是如果我们使用Spring的RmiServiceExporter把该类转变为RMI服务，那现有的实现不需要做任何改变。

RmiServiceExporter可以把任意Spring管理的bean发布为RMI服务。如图15.4所示，RmiServiceExporter把bean包装在一个适配器类中，然后适配器类被绑定到RMI注册表中，并且代理到服务类的请求——在本例中服务类也就是SpitterServiceImpl。

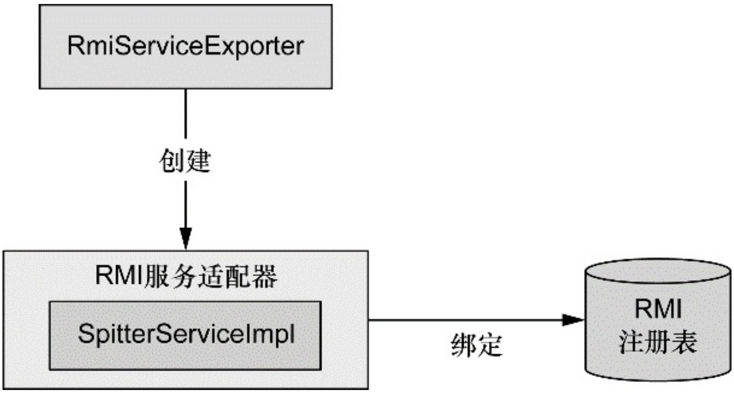

图15.4　RmiServiceExporter把POJO包装到服务适配器中，并将服务适配器绑定到RMI注册表中，从而将POJO转换为RMI服务

使用RmiServiceExporter将SpitterServiceImpl发布为RMI服务的最简单方式是在Spring中使用如下的@Bean方法进行配置：

```
@Bean
public RmiServiceExporter rmiExporter(SpitterService spitterService) {
  RmiServiceExporter rmiExporter = new RmiServiceExporter();
  rmiExporter.setService(spitterService);
  rmiExporter.setServiceName("SpitterService");
  rmiExporter.setServiceInterface(SpitterService.class);
  return rmiExporter;
}
```

这里会把spitterServicebean设置到service属性中，表明RmiServiceExporter要把该bean发布为一个RMI服务。serviceName属性命名了RMI服务，serviceInterface属性指定了此服务所实现的接口。

默认情况下，RmiServiceExporter会尝试绑定到本地机器1099端口上的RMI注册表。如果在这个端口没有发现RMI注册表，RmiServiceExporter将会启动一个注册表。如果希望绑定到不同端口或主机上的RMI注册表，那么我们可以通过registryPort和registryHost属性来指定。例如，下面的RmiServiceExporter会尝试绑定rmi.spitter.com主机1199端口上的RMI注册表：

```
@Bean
public RmiServiceExporter rmiExporter(SpitterService spitterService) {
  RmiServiceExporter rmiExporter = new RmiServiceExporter();
  rmiExporter.setService(spitterService);
  rmiExporter.setServiceName("SpitterService");
  rmiExporter.setServiceInterface(SpitterService.class);
  rmiExporter.setRegistryHost("rmi.spitter.com");
  rmiExporter.setRegistryPort(1199);
  return rmiExporter;
}
```

这就是我们使用Spring把某个bean转变为RMI服务所需要做的全部工作。现在Spitter服务已经导出为RMI服务，我们可以为Spittr应用创建其他的用户界面或邀请第三方使用此RMI服务创建新的客户端。如果使用Spring，客户端开发者访问Spitter的RMI服务会非常容易。

让我们转换一下视角来看看如何编写Spitter RMI服务的客户端。

15.2.2 装配RMI服务

传统上，RMI客户端必须使用RMI API的`Naming`类从RMI注册表中查找服务。例如，下面的代码片段演示了如何获取Spitter的RMI服务：

```
try {
  String serviceUrl = "rmi:/spitter/SpitterService";
  SpitterService spitterService =
          (SpitterService) Naming.lookup(serviceUrl);
  ...
}
catch (RemoteException e) { ... }
catch (NotBoundException e) { ... }
catch (MalformedURLException e) { ... }
```

虽然这段代码可以获取Spitter的RMI服务的引用，但是它存在两个问题：

- 传统的RMI查找可能会导致3种检查型异常的任意一种（`RemoteException`、`NotBoundException`和`MalformedURLException`），这些异常必须被捕获或重新抛出；
- 需要Spitter服务的任何代码都必须自己负责获取该服务。这属于样板代码，与客户端的功能并没有直接关系。

RMI查找过程中所抛出的异常通常意味着应用发生了致命的不可恢复的问题。例如，`MalformedURLException`异常意味着这个服务的地址是无效的。为了从这个异常中恢复，应用至少要重新配置，也可能需要重新编译。`try/catch`代码块并不能在发生异常时优雅地恢复，既然如此，为什么还要强制我们的代码捕获并处理这个异常呢？

但是，更糟糕的事情是这段代码直接违反了依赖注入（DI）原则。因为客户端代码需要负责查找Spitter服务，并且这个服务是RMI服务，我们甚至没有任何机会去提供`SpitterService`对象的不同实现。理想情况下，应该可以为任意一个bean注入`SpitterService`对象，而不是让bean自己去查找服务。利用DI，`SpitterService`的任何客户端都不需要关心此服务来源于何处。

Spring的RmiProxyFactoryBean是一个工厂bean，该bean可以为RMI服务创建代理。使用RmiProxyFactoryBean引用SpitterService的RMI服务是非常简单的，只需要在客户端的Spring配置中增加如下的@Bean方法：

```
@Bean
public RmiProxyFactoryBean spitterService() {
  RmiProxyFactoryBean rmiProxy = new RmiProxyFactoryBean();
  rmiProxy.setServiceUrl("rmi://localhost/SpitterService");
  rmiProxy.setServiceInterface(SpitterService.class);
  return rmiProxy;
}
```

服务的URL是通过RmiProxyFactoryBean的serviceUrl属性来设置的，在这里，服务名被设置为SpitterService，并且声明服务是在本地机器上的；同时，服务提供的接口由serviceInterface属性来指定。图15.5展示了客户端和RMI代理的交互。

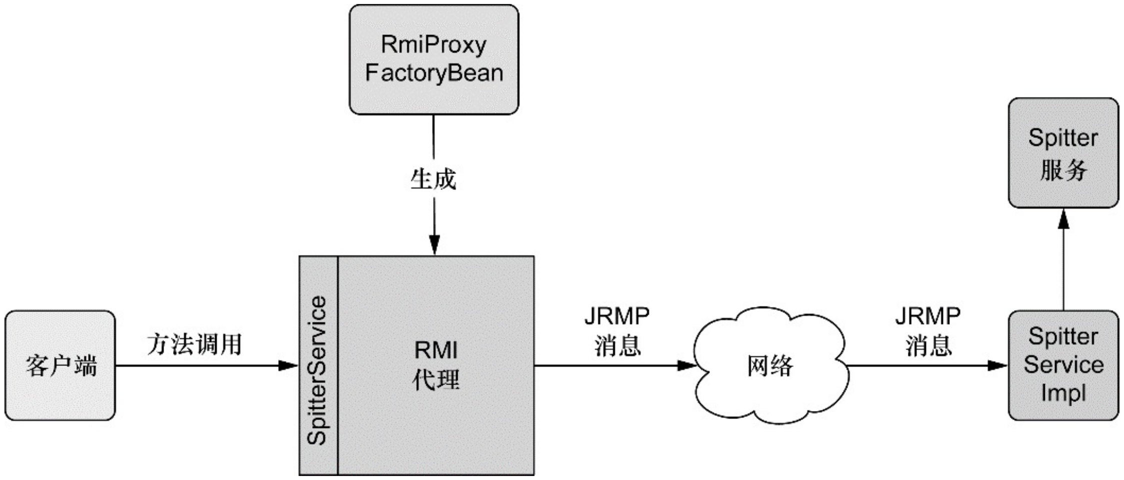

图15.5　RmiProxyFactoryBean生成一个代理对象，该对象代表客户端来负责与远程的RMI服务进行通信。客户端通过服务的接口与代理进行交互，就如同远程服务就是一个本地的POJO

现在已经把RMI服务声明为Spring管理的bean，我们就可以把它作为依赖装配进另一个bean中，就像任意非远程的bean那样。例如，假设客户端需要使用Spitter服务为指定的用户获取Spittle列表，我们可以使用@Autowired注解把服务代理装配进客户端中：

```
@Autowired
SpitterService spitterService;
```

我们还可以像本地bean一样调用它的方法：

```
public List<Spittle> getSpittles(String userName) {
  Spitter spitter = spitterService.getSpitter(userName);
  return spitterService.getSpittlesForSpitter(spitter);
}
```

以这种方式访问RMI服务简直太棒了！客户端代码甚至不需要知道所处理的是一个RMI服务。它只是通过注入机制接受了一个SpitterService对象，根本不必关心它来自何处。实际上，谁知道客户端得到的就是一个基于RMI的实现呢？

此外，代理捕获了这个服务所有可能抛出的RemoteException异常，并把它包装为运行期异常重新抛出，这样我们就可以放心地忽略这些异常。我们也可以非常容易地把远程服务bean替换为该服务的其他实现——或许是不同的远程服务，或者可能是客户端代码单元测试时的一个mock实现。

虽然客户端代码根本不需要关心所赋予的SpitterService是一个远程服务，但我们需要非常谨慎地设计远程服务的接口。提醒一下，客户端不得不调用两次服务：一次是根据用户名查找Spitter，另一次是获取Spittle对象的列表。这两次远程调用都会受网络延迟的影响，进而可能会影响到客户端的性能。清楚了客户端是如何使用服务的，我们或许会重写接口，把这两个调用放进一个方法中。但是现在我们要接受这样的服务接口。

RMI是一种实现远程服务交互的好办法，但是它存在某些限制。首先，RMI很难穿越防火墙，这是因为RMI使用任意端口来交互——这是防火墙通常所不允许的。在企业内部网络环境中，我们通常不需要担心这个问题。但是如果在互联网上运行，我们用RMI可能会遇到麻烦。即使RMI提供了对HTTP的通道的支持（通常防火墙都允许），但是建立这个通道也不是件容易的事。

另外一件需要考虑的事情是RMI是基于Java的。这意味着客户端和服务端必须都是用Java开发的。因为RMI使用了Java的序列化机制，所以通过网络传输的对象类型必须要保证在调用两端的Java运行时中是完全相同的版本。对我们的应用而言，这可能是个问题，也可能不是问题。但是选择RMI做远程服务时，必须要牢记这一点。

Caucho Technology（Resin应用服务器背后的公司）开发了一套应对RMI限制的远程调用解决方案。实际上，Caucho提供了两种解决方案：Hessian和Burlap。让我们看一下如何在Spring中使用Hessian和Burlap处理远程服务。

15.3 使用Hessian和Burlap发布远程服务

Hessian和Burlap是Caucho Technology提供的两种基于HTTP的轻量级远程服务解决方案。借助于尽可能简单的API和通信协议，它们都致力于简化Web服务。

你可能会好奇，为什么Caucho对同一个问题会有两种解决方案。Hessian和Burlap就如同一个事物的两面，但是每一个解决方案都服务于略微不同的目的。

Hessian，像RMI一样，使用二进制消息进行客户端和服务端的交互。但与其他二进制远程调用技术（例如RMI）不同的是，它的二进制消息可以移植到其他非Java的语言中，包括PHP、Python、C++和C#。

Burlap是一种基于XML的远程调用技术，这使得它可以自然而然地移植到任何能够解析XML的语言上。正因为它基于XML，所以相比起Hessian的二进制格式而言，Burlap可读性更强。但是和其他基于XML的远程技术（例如SOAP或XML-RPC）不同，Burlap的消息结构尽可能的简单，不需要额外的外部定义语言（例如WSDL或IDL）。

你可能想知道如何在Hessian和Burlap之间做出选择。很大程度上，它们是一样的。唯一的区别在于Hessian的消息是二进制的，而Burlap的消息是XML。由于Hessian的消息是二进制的，所以它在带宽上更具优势。但是如果我们更注重可读性（如出于调试的目的）或者我们的应用需要与没有Hessian实现的语言交互，那么Burlap的XML消息会是更好的选择。

为了在Spring中演示Hessian和Burlap服务，让我们回顾一下在前一节中使用RMI解决Spitter服务的示例。但是这一次，我们将看看如何使用Hessian和Burlap作为远程调用模型来解决这个问题。

15.3.1 使用Hessian和Burlap导出bean的功能

像之前一样，我们希望把SpitterServiceImpl类的功能发布为远程服务——这次是一个Hessian服务。即使没有Spring，编写一个Hessian服务也是相当容易的。我们只需要编写一个继承`com.caucho.hessian.server.HessianServlet`的类，并确保所有的服务方法是`public`的（在Hessian里，所有public方法被视为服务方法）。

因为Hessian服务很容易实现，Spring并没有做更多简化Hessian模型的工作。但是和Spring一起使用时，Hessian服务可以在各方面利用Spring框架的优势，这是纯Hessian服务所不具备的。包括利用Spring的AOP来为Hessian服务提供系统级服务，例如声明式事务。

导出**Hessian**服务

在Spring中导出一个Hessian服务和在Spring中实现一个RMI服务惊人的相似。为了把Spitter服务bean发布为RMI服务，我们需要在Spring配置文件中配置一个`RmiServiceExporter`bean。同样的方式，为了把Spitter服务发布为Hessian服务，我们需要配置另一个导出bean，只不过这次是`HessianServiceExporter`。

`HessianServiceExporter`对Hessian服务所执行的功能与`RmiServiceExporter`对RMI服务所执行的功能是相同的：它把POJO的public方法发布成Hessian服务的方法。不过，正如图15.6所示，其实现过程与`RmiServiceExporter`将POJO发布为RMI服务是不同的。

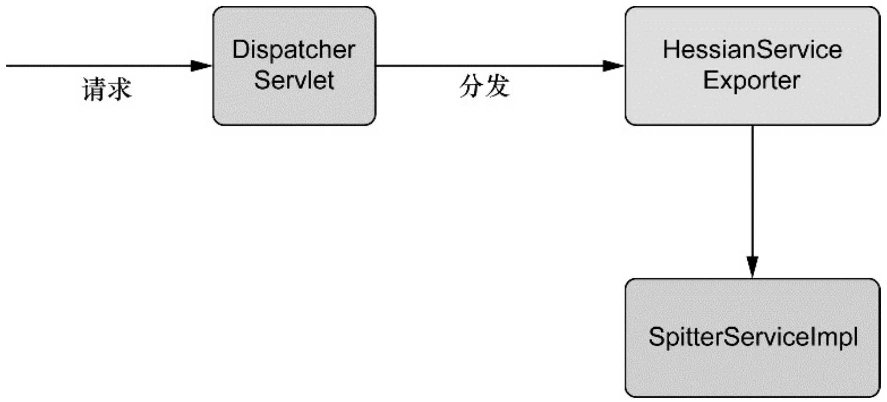

图15.6　HessianServiceExporter是一个Spring MVC控制器，它可以接收Hessian请求，并把这些请求转换成对POJO的调用从而将POJO导出为一个Hessian服务

`HessianServiceExporter`（稍后会有更详细的介绍）是一个Spring MVC控制器，它接收Hessian请求，并将这些请求转换成对被导出POJO的方法调用。在如下Spring的声明中，`HessianServiceExporter`会把`spitterService` bean导出为Hessian服务：

```
@Bean
public HessianServiceExporter
      hessianExportedSpitterService(SpitterService service) {
  HessianServiceExporter exporter = new HessianServiceExporter();
  exporter.setService(service);
  exporter.setServiceInterface(SpitterService.class);
  return exporter;
}
```

正如RmiServiceExporter一样，service属性的值被设置为实现了这个服务的bean引用。在这里，它引用的是spitterServicebean。serviceInterface属性用来标识这个服务实现了SpitterService接口。

与RmiServiceExporter不同的是，我们不需要设置serviceName属性。在RMI中，serviceName属性用来在RMI注册表中注册一个服务。而Hessian没有注册表，因此也就没必要为Hessian服务进行命名。

配置**Hessian**控制器

RmiServiceExporter和HessianServiceExporter另外一个主要区别就是，由于Hessian是基于HTTP的，所以HessianSeriviceExporter实现为一个Spring MVC控制器。这意味着为了使用导出的Hessian服务，我们需要执行两个额外的配置步骤：

- 在web.xml中配置Spring的DispatcherServlet，并把我们的应用部署为 Web应用；
- 在Spring的配置文件中配置一个URL处理器，把Hessian服务的URL分发给对应的Hessian服务bean。

我们在第5章学习了如何配置Spring的DispatcherServlet和URL处理器，所以这些步骤看起来有些熟悉。首先，我们需要一个DispatcherServlet。还好，这个我们已经在Spittr应用的web.xml文件中配置了。但是为了处理Hessian服务，DispatcherServlet还需要配置一个Servlet映射来拦截后缀为“*.service”的URL：

```
<servlet-mapping>
  <servlet-name>spitter</servlet-name>
  <url-pattern>*.service</url-pattern>
</servlet-mapping>
```

如果你在Java中通过实现WebApplicationInitializer来配置DispatcherServlet的话，那么需要将URL模式作为映射添加

到ServletRegistration.Dynamic中，在将DispatcherServlet添加到容器中的时候，我们能够得到ServletRegistration.Dynamic对象：

```
ServletRegistration.Dynamic dispatcher = container.addServlet(
        "appServlet", new DispatcherServlet(dispatcherServletContext));
    dispatcher.setLoadOnStartup(1);
    dispatcher.addMapping("/");
    dispatcher.addMapping("*.service");
```

或者，如果你通过扩展AbstractDispatcherServletInitializer或Abstract-AnnotationConfigDispatcherServletInitializer的方式来配置DispatcherServlet，那么在重载getServletMappings()的时候，需要包含该映射：

```
@Override
  protected String[] getServletMappings() {
    return new String[] { "/", "*.service" };
  }
```

这样配置后，任何以“.service”结束的URL请求都将由DispatcherServlet处理，它会把请求传递给匹配这个URL的控制器。因此“/spitter.service”的请求最终将被hessianSpitterServicebean所处理（它实际上仅仅是一个SpitterServiceImpl的代理）。

那我们是如何知道这个请求会转给hessianSpitterSevice处理呢？我们还需要配置一个URL映射来确保DispatcherServlet把请求转给hessianSpitterService。如下的SimpleUrlHandlerMappingbean可以做到这一点：

```
@Bean
public HandlerMapping hessianMapping() {
  SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
  Properties mappings = new Properties();
  mappings.setProperty("/spitter.service",
                       "hessianExportedSpitterService");
  mapping.setMappings(mappings);
  return mapping;
}
```

如果不喜欢Hessian的二进制协议，我们还可以选择使用Burlap基于XML的协议。让我们看看如何把一个服务导出为Burlap服务。

导出**Burlap**服务

从任何方面上看，BurlapServiceExporter

与HessianServiceExporter实际上都是相同的，只不过它使用基于XML的协议而不是二进制协议。下面的bean定义展示了如何使用BurlapServiceExporter把Spitter服务导出为一个Burlap服务：

```
@Bean
public BurlapServiceExporter
      burlapExportedSpitterService(SpitterService service) {
  BurlapServiceExporter exporter = new BurlapServiceExporter();
  exporter.setService(service);
  exporter.setServiceInterface(SpitterService.class);
  return exporter;
}
```

正如我们所看到的，这个bean与使用Hessian所对应bean的唯一区别在于bean的方法和导出类。配置Burlap服务和配置Hessian服务是一模一样的，这包括需要准备一个URL处理器和一个DispatcherServlet。

现在让我们看看会话的另一端，如何访问我们使用Hessian（或Burlap)所发布的服务。

15.3.2 访问Hessian/Burlap服务

回顾一下在15.2.2小节中，在使用RmiProxyFactoryBean访问Spitter服务的客户端代码中，完全不知道这个服务是一个RMI服务。事实上，也根本没有任何迹象表明这个服务是一个远程服务。它只是与SpitterService接口打交道——RMI的所有细节完全包含在Spring配置中这个bean的配置中。好处是客户端不需要了解服务的实现，因此从RMI客户端转到Hessian客户端会变得极其简单，不需要改变任何客户端的Java代码。

坏处是，如果你真的喜欢编写Java代码的话，那么这一节或许让你大失所望。这是因为在客户端代码中，基于RMI的服务与基于Hessian的服务之间唯一的差别在于要使用Spring的HessianProxyFactoryBean来代替RmiProxyFactoryBean。客户端调用基于Hessian的Spitter服务可以用如下的配置声明：

```
@Bean
public HessianProxyFactoryBean spitterService() {
  HessianProxyFactoryBean proxy = new HessianProxyFactoryBean();
  proxy.setServiceUrl("http://localhost:8080/Spitter/spitter.service");
  proxy.setServiceInterface(SpitterService.class);
  return proxy;
}
```

就像基于RMI服务那样，serviceInterface属性指定了这个服务实现的接口。并且，像RmiProxyFactoryBean一

样，serviceUrl标识了这个服务的URL。既然Hessian是基于HTTP的，当然我们在这里要设置一个HTTP URL（URL是由我们先前定义的URL映射所决定的）。图15.7展示了客户端以及由HessianProxyFactoryBean所生成的代理之间是如何交互的。

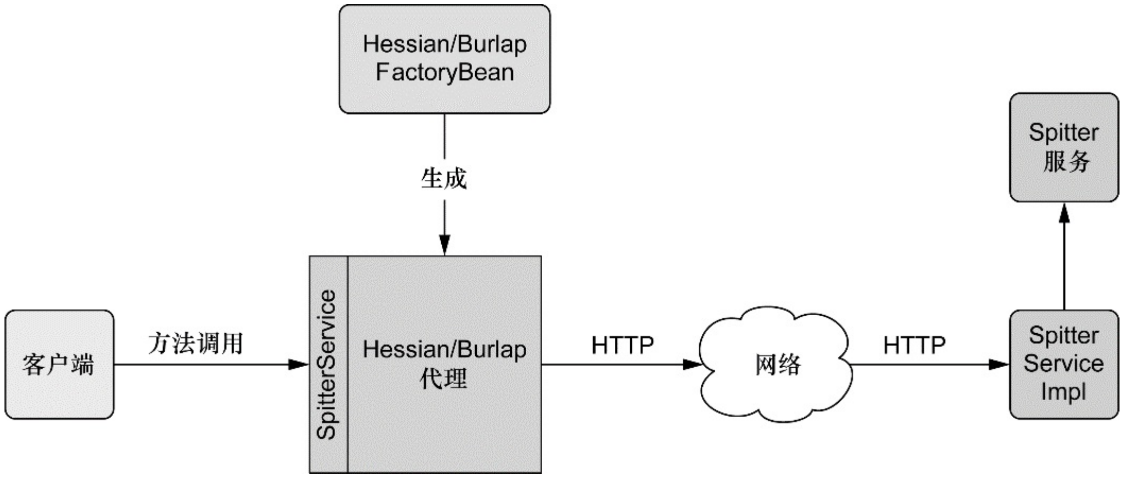

图15.7　HessianProxyFactoryBean和BurlapProxyFactoryBean生成的代理对象负责通过HTTP（Hessian为二进制、Burlap为XML）与远程对象通信

事实证明，把Burlap服务装配进客户端同样也没有太多新意。二者唯一的区别在于，我们要使用BurlapProxyFactoryBean来代替HessianProxyFactoryBean：

```
@Bean
public BurlapProxyFactoryBean spitterService() {
  BurlapProxyFactoryBean proxy = new BurlapProxyFactoryBean();
  proxy.setServiceUrl("http://localhost:8080/Spitter/spitter.service");
  proxy.setServiceInterface(SpitterService.class);
  return proxy;
}
```

尽管我们觉得在RMI、Hessian和Burlap服务之间稍微不同的配置是很无趣的，但是这样的单调恰恰是有好处的。它意味着我们可以很容易在各种Spring所支持的远程调用技术之间进行切换，而不需要重新学习一个全新的模型。一旦我们配置了对RMI服务的引用，把它重新配置为Hessian或Burlap服务也是很轻松的工作。

因为Hessian和Burlap都是基于HTTP的，它们都解决了RMI所头疼的防火墙渗透问题。但是当传递过来的RPC消息中包含序列化对象时，RMI就完胜Hessian和Burlap了。因为Hessian和Burlap都采用了私有的序列化机制，而RMI使用的是Java本身的序列化机制。如果我们的数据模型非常复杂，Hessian/Burlap的序列化模型就可能　　　无法胜任了。

我们还有一个两全其美的解决方案。让我们看一下Spring的HTTP invoker，它基于HTTP提供了RPC（像Hessian/Burlap一样），同时又使用了Java的对象序列化机制（像RMI一样）。

15.4 使用Spring的HttpInvoker

Spring开发团队意识到RMI服务和基于HTTP的服务（例如Hessian和Burlap）之间的空白。一方面，RMI使用Java标准的对象序列化机制，但是很难穿透防火墙。另一方面，Hessian和Burlap能很好地穿透防火墙，但是使用私有的对象序列化机制。

就这样，Spring的HTTP invoker应运而生了。HTTP invoker是一个新的远程调用模型，作为Spring框架的一部分，能够执行基于HTTP的远程调用（让防火墙不为难），并使用Java的序列化机制（让开发者也乐观其变）。使用基于HTTP invoker的服务和使用基于Hessian/Burlap的服务非常相似。

为了开始学习HTTP invoker，让我们再来看一下Spitter服务——这一次我们将作为HTTP invoker服务来实现。

15.4.1 将bean导出为HTTP服务

要将bean导出为RMI服务，我们需要使用`RmiServiceExporter`；要将bean导出为Hessian服务，我们需要使用`HessianServiceExporter`；要将bean导出为Burlap服务，我们需要使用`BurlapServiceExporter`。把这种千篇一律的用法带到HTTP invoker上，应该也不会有任何意外的事情发生，那就是导出HTTP invoker服务，我们需要使用`HttpInvokerServiceExporter`。

为了把Spitter服务导出为一个基于HTTP invoker的服务，我们需要像下面的配置一样声明一个`HttpInvokerServiceExporter`bean：

```
@Bean
public HttpInvokerServiceExporter
      httpExportedSpitterService(SpitterService service) {
  HttpInvokerServiceExporter exporter =
          new HttpInvokerServiceExporter();
  exporter.setService(service);
  exporter.setServiceInterface(SpitterService.class);
  return exporter;
}
```

是否有点似曾相识的感觉？我们很难找出这个bean的定义和那些在15.3.2小节中所声明的bean有什么不同。唯一的区别在于类名：HttpInvokerServiceExporter。否则的话，这个导出器和其他的远程服务的导出器就没有任何区别了。

如图15.8所示，HttpInvokerServiceExporter的工作方式与HessianService-Exporter和BurlapServiceExporter很相似。HttpInvokerServiceExporter也是一个Spring的MVC控制器，它通过DispatcherServlet接收来自于客户端的请求，并将这些请求转换成对实现服务的POJO的方法调用。

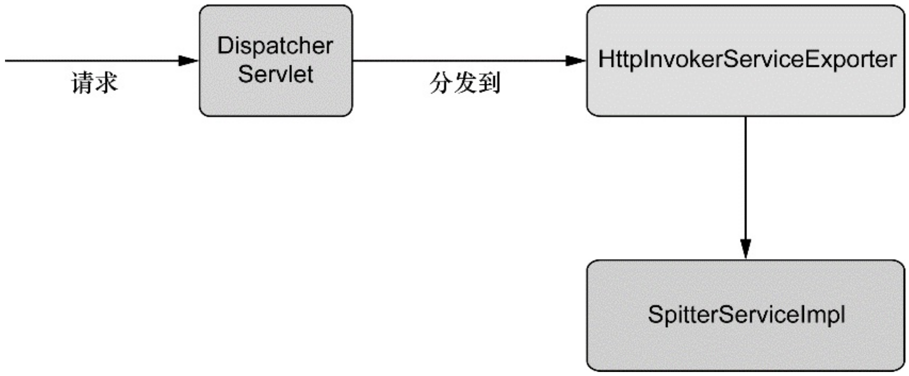

图15.8 HttpInvokerServiceExporter工作方式与Hessian和Burlap很相似，通过Spring MVC的DispatcherServlet接收请求，并将这些请求转换成对Spring bean的方法调用

因为HttpInvokerServiceExporter是一个Spring MVC控制器，我们需要建立一个URL处理器，映射HTTP URL到对应的服务上，就像Hessian和Burlap导出器所做的一样：

```
@Bean
public HandlerMapping httpInvokerMapping() {
  SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
  Properties mappings = new Properties();
  mappings.setProperty("/spitter.service",
                       "httpExportedSpitterService");
  mapping.setMappings(mappings);
  return mapping;
}
```

同样，像之前一样，我们需要确保匹配了DispatcherServlet，这样才能处理对“*.service”扩展的请求。参考15.3.1小节了解如何设置映射。

我们已经知道如何访问由RMI、Hessian或Burlap所创建的远程服务，现在我们再次让Spitter客户端使用刚才所导出的基于HTTP invoker的服务。

15.4.2 通过HTTP访问服务

这听起来像打破记录，但是我还得告诉你，访问基于HTTP invoker的服务很类似于我们之前使用的其他远程服务代理。实际上就是一样的。如图15.9所示，`HttpInvokerProxyFactoryBean`填充了相同的位置，正如我们在本章所看到的其他远程服务代理工厂bean一样。

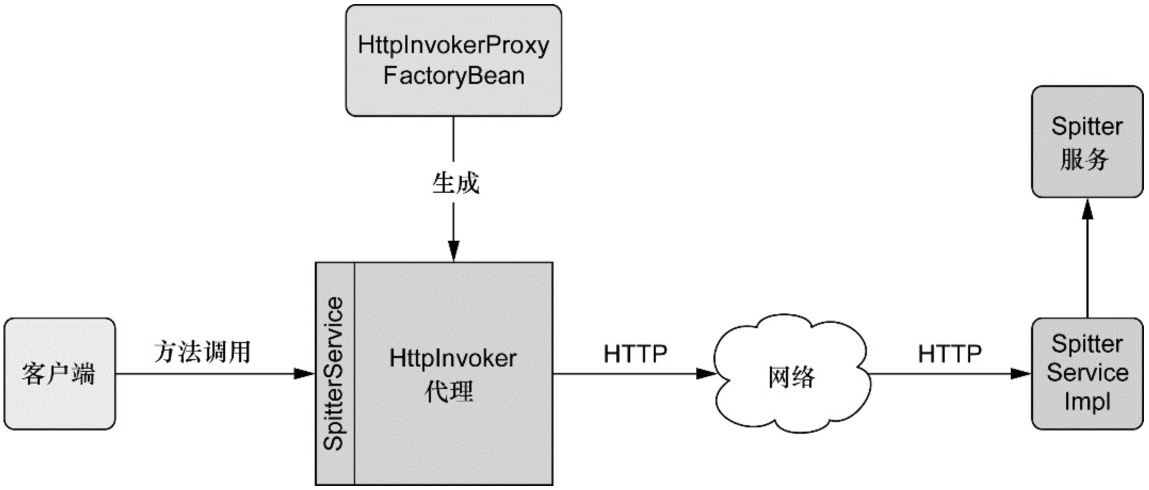

图15.9 HttpInvokerProxyFactoryBean是一个代理工厂bean，用于生成一个代理，该代理使用Spring特有的基于HTTP协议进行远程通信

为了把基于HTTP invoker的远程服务装配进我们的客户端Spring应用上下文中，我们必须将`HttpInvokerProxyFactoryBean`配置为一个bean来代理它，如下所示：

```
@Bean
public HttpInvokerProxyFactoryBean spitterService() {
  HttpInvokerProxyFactoryBean proxy = new HttpInvokerProxyFactoryBean();
  proxy.setServiceUrl("http://localhost:8080/Spitter/spitter.service");
  proxy.setServiceInterface(SpitterService.class);
  return proxy;
}
```

与15.2.2小节和15.3.2小节的bean定义相对比，我们会发现几乎没什么变化。`serviceInterface`属性仍然用来标识Spitter服务所实现的接口，而`serviceUrl`属性仍然用来标识远程服务的位置。因为HTTP invoker是基于HTTP的，如同Hessian和Burlap一样，`serviceUrl`可以包含与Hessian和Burlap版本中的bean一样的URL。

难道你不喜欢对称美吗？

Spring的HTTP invoker是作为两全其美的远程调用解决方案而出现

的，把HTTP的简单性和Java内置的对象序列化机制融合在一起。这使得HTTP invoker服务成为一个引人注目的替代RMI或Hessian/Burlap的可选方案。

要记住HTTP invoker有一个重大的限制：它只是一个Spring框架所提供的远程调用解决方案。这意味着客户端和服务端必须都是Spring应用。并且，至少目前而言，也隐含表明客户端和服务端必须是基于Java的。另外，因为使用了Java的序列化机制，客户端和服务端必须使用相同版本的类（与RMI类似）。

RMI、Hessian、Burlap和HTTP invoker都是远程调用的可选解决方案。但是当面临无所不在的远程调用时，Web服务是势不可挡的。下一节，我们将了解Spring如何对基于SOAP的Web服务远程调用提供支持。

15.5　发布和使用Web服务

近几年，最流行的一个TLA（三个字母缩写）就是SOA（面向服务的架构）。SOA对不同的人意味着不同的意义。但是，SOA的核心理念是，应用程序可以并且应该被设计成依赖于一组公共的核心服务，而不是为每个应用都重新实现相同的功能。

例如，一个金融机构可能有若干个应用，其中很多都需要访问借款者的账户信息。在这种情况下，应用应该都依赖于一个公共的获取账户信息的服务，而不应该在每一个应用中都建立账户访问逻辑（其中大部分逻辑都是重复的）。

Java与Web服务的结合已经有很长的历史了，而且在Java中使用Web服务有多种选择。其中的大多数可选方案已经以某种方式与Spring进行了整合。虽然Spring为使用Java API for XML Web Service（JAX-WS）来发布和使用SOAP Web服务提供了大力支持，但是在本书我不可能涵盖每一个Spring所支持的Web服务框架和工具箱。

在本节，我们重新回顾下Spitter服务示例，不过这次我们将使用Spring对JAX-WS的支持来把Spitter服务发布为Web服务并使用此Web服务。首先，我们来看一下如何在Spring中创建JAX-WS Web服务。

15.5.1　创建基于Spring的JAX-WS端点

在本章前面的内容中，我们使用Spring的服务导出器创建了远程服

务。这些服务导出器很神奇地将Spring配置的POJO转换成了远程服务。我们看到了如何使用RmiServiceExporter创建RMI服务，如何使用HessianServiceExporter创建Hessian服务，如何使用BurlapServiceExporter创建Burlap服务，以及如何使用HttpInvokerServiceExporter创建HTTP invoker服务。现在你或许期望我在本节展示如何使用一个JAX-WS服务导出器创建Web服务。

Spring的确提供了一个JAX-WS服务导出器，SimpleJaxWsServiceExporter，我们很快就可以看到。但在这之前，你必须知道它并不一定是所有场景下的最好选择。你是知道的，SimpleJaxWsServiceExporter要求JAX-WS运行时支持将端点发布到指定地址上。Sun JDK 1.6自带的JAX-WS可以符合要求，但是其他的JAX-WS实现，包括JAX-WS的参考实现，可能并不能满足此需求。

如果我们将要部署的JAX-WS运行时不支持将其发布到指定地址上，那我们就要以更为传统的方式来编写JAX-WS端点。这意味着端点的生命周期由JAX-WS运行时来进行管理，而不是Spring。但是这并不意味着它们不能装配Spring上下文中的bean。

在**Spring**中自动装配**JAX-WS**端点

JAX-WS编程模型使用注解将类和类的方法声明为Web服务的操作。使用@WebService注解所标注的类被认为Web服务的端点，而使用@WebMethod注解所标注的方法被认为是操作。

就像大规模应用中的其他对象一样，JAX-WS端点很可能需要与其他对象交互来完成工作。这意味着JAX-WS端点可以受益于依赖注入。但是如果端点的生命周期由JAX-WS运行时来管理，而不是由Spring来管理的话，这似乎不可能把Spring管理的bean装配进JAX-WS管理的端点实例中。

装配JAX-WS端点的秘密在于继承SpringBeanAutowiringSupport。通过继承SpringBeanAutowiringSupport，我们可以使用@Autowired注解标注端点的属性，依赖就会自动注入了。SpitterServiceEndpoint展示了它是如何工作的。

程序清单**15.2　JAX-WS**端点中的**SpitterBeanAutowiringSupport**

```
package com.habuma.spittr.remoting.jaxws;
import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebService;
import org.springframework.beans.factory.annotation.Autowired;
import
  org.springframework.web.context.support.SpringBeanAutowiringSupport;
import com.habuma.spittr.domain.Spitter;
import com.habuma.spittr.domain.Spittle;
import com.habuma.spittr.service.SpitterService;
@WebService(serviceName="SpitterService")
public class SpitterServiceEndpoint
    extends SpringBeanAutowiringSupport {          <-- 启用自动装配
   @Autowired
  SpitterService spitterService;             <-- 自动装配 SpitterService
   @WebMethod
  public void addSpittle(Spittle spittle) {
    spitterService.saveSpittle(spittle);          <--
    }                                              委托给
  @WebMethod                                       SpitterService
  public void deleteSpittle(long spittleId) {
    spitterService.deleteSpittle(spittleId);      <--
    }
  @WebMethod
  public List<Spittle> getRecentSpittles(int spittleCount) {
    return spitterService.getRecentSpittles(spittleCount);    <--
    }                                                          委托给
  @WebMethod                                                   SpitterService
  public List<Spittle> getSpittlesForSpitter(Spitter spitter) {
    return spitterService.getSpittlesForSpitter(spitter);     <--
    }
}
```

我们在`SpitterService`属性上使用`@Autowired`注解来表明它应该自动注入一个从Spring应用上下文中所获取的bean。在这里，端点委托注入的`SpitterService`来完成实际的工作。

导出独立的JAX-WS端点

正如我所说的，当对象的生命周期不是由Spring管理的，而对象的属性又需要注入Spring所管理的bean时，`SpringBeanAutowiringSupport`很有用。在合适场景下，还是可以把Spring管理的bean导出为JAX-WS端点的。

`SpringSimpleJaxWsServiceExporter`的工作方式很类似于本章前边所介绍的其他服务导出器。它把Spring管理的bean发布为JAX-WS运行时中的服务端点。与其他服务导出器不同，`SimpleJaxWsServiceExporter`不需要为它指定一个被导出bean的引用，它会将使用JAX-WS注解所标注的所有bean发布为JAX-WS服务。

`SimpleJaxWsServiceExporter`可以使用如下的`@Bean`方法来配置：

```
@Bean
public SimpleJaxWsServiceExporter jaxWsExporter() {
  return new SimpleJaxWsServiceExporter();
}
```

正如我们所看到的，SimpleJaxWsServiceExporter不需要再做其他的事情就可以完成所有的工作。当启动的时候，它会搜索Spring应用上下文来查找所有使用@WebService注解的bean。当找到符合的bean时，SimpleJaxWsServiceExporter使用http://localhost:8080/ 地址将bean发布为JAX-WS端点。SpitterServiceEndpoint就是其中一个被查找到的bean。

程序清单15.3 SimpleJaxWsServiceExporter将bean转变为JAX-WS端点

```
package com.habuma.spittr.remoting.jaxws;
import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.habuma.spittr.domain.Spitter;
import com.habuma.spittr.domain.Spittle;
import com.habuma.spittr.service.SpitterService;
@Component
@WebService(serviceName="SpitterService")
public class SpitterServiceEndpoint {
  @Autowired
  SpitterService spitterService;                    <— 自动装配 SpitterService
   @WebMethod
  public void addSpittle(Spittle spittle) {
    spitterService.saveSpittle(spittle);                <— 委托给 SpitterService
   }
  @WebMethod
  public void deleteSpittle(long spittleId) {
    spitterService.deleteSpittle(spittleId);                <—┐
   }                                                            │
  @WebMethod                                                    │  委托给
  public List<Spittle> getRecentSpittles(int spittleCount) {    │  SpitterService
    return spitterService.getRecentSpittles(spittleCount);  <—┤
   }                                                            │
  @WebMethod                                                    │
  public List<Spittle> getSpittlesForSpitter(Spitter spitter) { │
    return spitterService.getSpittlesForSpitter(spitter);   <—┘
   }
}
```

我们注意到SpitterServiceEndpoint的新实现不再继承SpringBeanAutowiring-Support了。它完全就是一个Spring bean，因此SpitterServiceEndpoint不需要继承任何特殊的支持类就可以实现自动装配。

因为SimpleJaxWsServiceEndpoint的默认基本地址为http://localhost:8080/，而SpitterServiceEndpoint使用了

@Webservice(servicename="SpitterService")注解，所以这两个bean所形成的Web服务地址均为http://localhost:8080/SpitterService。但是我们可以完全控制服务URL，如果希望调整服务URL的话，我们可以调整基本地址。例如，如下SimpleJaxWsServiceEndpoint的配置把相同的服务端点发布到http://localhost:8888 /srvices/SpitterService。

```
@Bean
public SimpleJaxWsServiceExporter jaxWsExporter() {
  SimpleJaxWsServiceExporter exporter =
          new SimpleJaxWsServiceExporter();
  exporter.setBaseAddress("http://localhost:8888/services/");
}
```

SimpleJaxWsServiceEndpoint就像看起来那么简单，但是我们应该注意它只能用在支持将端点发布到指定地址的JAX-WS运行时中。这包含了Sun 1.6 JDK自带的JAX-WS运行时。其他的JAX-WS运行时，例如JAX-WS 2.1的参考实现，不支持这种类型的端点发布，因此也就不能使用SimpleJaxWsServiceEndpoint。

15.5.2 在客户端代理JAX-WS服务

使用Spring发布Web服务与我们使用RMI、Hessian、Burlap和HTTP invoker发布服务是有所不同的。但是我们很快就会发现，借助Spring使用Web服务所涉及的客户端代理的工作方式与基于Spring的客户端使用其他远程调用技术是相同的。

使用JaxWsProxyFactoryBean，我们可以在Spring中装配Spitter Web服务，与任意一个其他的bean一样。JaxWsProxyFactoryBean是Spring工厂bean，它能生成一个知道如何与SOAP Web服务交互的代理。所创建的代理实现了服务接口（如图15.10所示）。因此，JaxWsProxyFactoryBean让装配和使用一个远程Web服务变成了可能，就像这个远程Web服务是本地POJO一样。

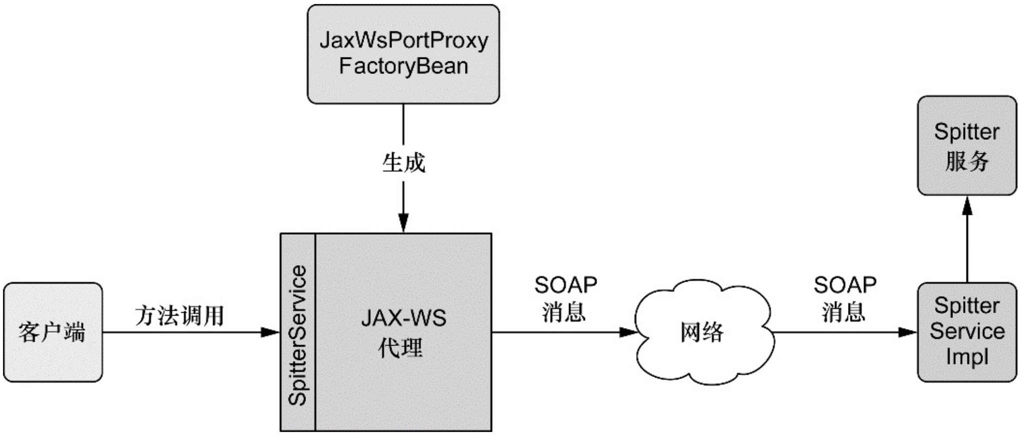

图15.10　JaxWsPortProxyFactoryBean生成可以与远程Web服务交互的代理。这些代理可以被装配到其他bean中，就像它们是本地POJO一样

我们可以像下面这样配置JaxWsPortProxyFactoryBean来引用Spitter服务：

```
@Bean
public JaxWsPortProxyFactoryBean spitterService() {
  JaxWsPortProxyFactoryBean proxy = new JaxWsPortProxyFactoryBean();
  proxy.setWsdlDocument(
          "http://localhost:8080/services/SpitterService?wsdl");
  proxy.setServiceName("spitterService");
  proxy.setPortName("spitterServiceHttpPort");
  proxy.setServiceInterface(SpitterService.class);
  proxy.setNamespaceUri("http://spitter.com");
  return proxy;
}
```

我们可以看到，为JaxWsPortProxyFactoryBean设置几个属性就可以工作了。wsdlDocumentUrl属性标识了远程Web服务定义文件的位置。JaxWsPortProxyFactory bean将使用这个位置上可用的WSDL来为服务创建代理。由JaxWsPortProxyFactoryBean所生成的代理实现了serviceInterface属性所指定的SpitterService接口。

剩下的三个属性的值通常可以通过查看服务的WSDL来确定。为了演示，我们假设为Spitter服务的WSDL如下所示：

```
<wsdl:definitions targetNamespace="http://spitter.com">
...
  <wsdl:service name="spitterService">
    <wsdl:port name="spitterServiceHttpPort"
            binding="tns:spitterServiceHttpBinding">
...
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

虽然不太可能这么做，但是在服务的WSDL中定义多个服务和端口是

允许的。鉴于此，JaxWsPortProxyFactoryBean需要我们使用portName和serviceName属性指定端口和服务名称。WSDL中<wsdl:port>和<wsdl:service>元素的name属性可以帮助我们识别出这些属性该设置成什么。

最后，namespaceUri属性指定了服务的命名空间。命名空间将有助于JaxWsPortProxyFactoryBean去定位WSDL中的服务定义。正如端口和服务名一样，我们可以在WSDL中找到该属性的正确值。它通常会在<wsdl:definitions>的targetNamespace属性中。

15.6　小结

使用远程服务通常是一个乏味的苦差事，但是Spring提供了对远程服务的支持，让使用远程服务与使用普通的JavaBean一样简单。

在客户端，Spring提供了代理工厂bean，能让我们在Spring应用中配置远程服务。不管是使用RMI、Hessian、Burlap、Spring的HTTP invoker，还是Web服务，都可以把远程服务装配进我们的应用中，好像它们就是POJO一样。Spring甚至捕获了所有的RemoteExecption异常，并在发生异常的地方重新抛出运行期异常RemoteAccessException，让我们的代码可以从处理不可恢复的异常中解放出来。

即便Spring隐藏了远程服务的很多细节，让它们表现得好像是本地JavaBean一样，但是我们应该时刻谨记它们是远程服务的事实。远程服务，本质上来讲，通常比本地服务更低效。当编写访问远程服务的代码时，我们必须考虑到这一点，限制远程调用，以规避性能瓶颈。

在本章，我们看到了Spring是如何使用几种基本的远程调用技术来发布和使用服务的。尽管这些远程调用方案在分布式应用中很有价值，但这只是涉及面向服务架构（SOA）的一鳞半爪。

我们还了解了如何将bean导出为基于SOAP的Web服务。尽管这是开发Web服务的一种简单方式，但从架构角度来看，它可能不是最佳的选择。在下一章，我们将学习构建分布式应用的另一种选择，把应用暴露为RESTful资源。

第16章　使用Spring MVC创建REST API

本章内容：

- 编写处理REST资源的控制器
- 以XML、JSON及其他格式来表述资源
- 使用REST资源

数据为王。

作为开发人员，我们经常关注于构建伟大的软件来解决业务问题。数据只是软件完成工作时要处理的原材料。但是如果你问一下业务人员，数据和软件谁更重要的话，他们很可能会选择数据。数据是许多业务的生命之血。软件通常是可以替换的，但是多年积累的数据是永远不能替换的。

你是不是觉得有些奇怪，既然数据如此重要，为何在开发软件的时候却经常将其视为事后才考虑的事情？以我们前面上一章所介绍的远程服务为例，这些服务是以操作和处理为中心的，而不是信息和资源。

近几年来，以信息为中心的表述性状态转移（Representational State Transfer，REST）已成为替换传统SOAP Web服务的流行方案。SOAP一般会关注行为和处理，而REST关注的是要处理的数据。

从Spring 3.0版本开始，Spring为创建REST API提供了良好的支持。Spring的REST实现在Spring 3.1、3.2和如今的4.0版本中不断得到发展。

好消息是Spring对REST的支持是构建在Spring MVC之上的，所以我们已经了解了许多在Spring中使用REST所需的知识。在本章中，我们将基于已了解的Spring MVC知识来开发处理RESTful资源的控制器。但在深入了解细节之前，先让我们看看使用REST到底是什么。

16.1　了解REST

我敢打赌这并不是你第一次听到或读到REST这个词。近些年来，关

于REST已经有了许多讨论，在软件开发中你可能会发现有一种很流行的做法，那就是在推动REST替换SOAP Web服务的时候，会谈论到SOAP的不足。

诚然，对于许多应用程序而言，使用SOAP可能会有些大材小用了，而REST提供了一个更简单的可选方案。另外，很多的现代化应用都会有移动或富JavaScript客户端，它们都会使用运行在服务器上REST API。

问题在于并不是每个人都清楚REST到底是什么。结果就出现了许多误解。有很多打着REST幌子的事情其实并不符合REST真正的本意。在谈论Spring如何支持REST之前，我们需要对REST是什么达成共识。

16.1.1　REST的基础知识

当谈论REST时，有一种常见的错误就是将其视为“基于URL的Web服务”——将REST作为另一种类型的远程过程调用（remote procedure call，RPC）机制，就像SOAP一样，只不过是通过简单的HTTP URL来触发，而不是使用SOAP大量的XML命名空间。

恰好相反，REST与RPC几乎没有任何关系。RPC是面向服务的，并关注于行为和动作；而REST是面向资源的，强调描述应用程序的事物和名词。

为了理解REST是什么，我们将它的首字母缩写拆分为不同的构成部分：

- 表述性（*Representational*）：REST资源实际上可以用各种形式来进行表述，包括XML、JSON（JavaScript Object Notation）甚至HTML——最适合资源使用者的任意形式；
- 状态（*State*）：当使用REST的时候，我们更关注资源的状态而不是对资源采取的行为；
- 转移（*Transfer*）：REST涉及到转移资源数据，它以某种表述性形式从一个应用转移到另一个应用。

更简洁地讲，REST就是将资源的状态以最适合客户端或服务端的形式从服务器端转移到客户端（或者反过来）。

在REST中，资源通过URL进行识别和定位。至于RESTful URL的结构并没有严格的规则，但是URL应该能够识别资源，而不是简单的发一

条命令到服务器上。再次强调，关注的核心是事物，而不是行为。

REST中会有行为，它们是通过HTTP方法来定义的。具体来讲，也就是GET、POST、PUT、DELETE、PATCH以及其他的HTTP方法构成了REST中的动作。这些HTTP方法通常会匹配为如下的CRUD动作：

- Create：POST
- Read：GET
- Update：PUT或PATCH
- Delete：DELETE

尽管通常来讲，HTTP方法会映射为CRUD动作，但这并不是严格的限制。有时候，PUT可以用来创建新资源，POST可以用来更新资源。实际上，POST请求非幂等性（non-idempotent）的特点使其成为一个非常灵活的方法，对于无法适应其他HTTP方法语义的操作，它都能够胜任。

基于对REST的这种观点，所以我尽量避免使用诸如REST服务、REST Web服务或类似的术语，这些术语会不恰当地强调行为。相反，我更愿意强调REST面向资源的本质，并讨论RESTful资源。

16.1.2 Spring是如何支持REST的

Spring很早就有导出REST资源的需求。从3.0版本开始，Spring针对Spring MVC的一些增强功能对REST提供了良好的支持。当前的4.0版本中，Spring支持以下方式来创建REST资源：

- 控制器可以处理所有的HTTP方法，包含四个主要的REST方法：GET、PUT、DELETE以及POST。Spring 3.2及以上版本还支持PATCH方法；
- 借助@PathVariable注解，控制器能够处理参数化的URL（将变量输入作为URL的一部分）；
- 借助Spring的视图和视图解析器，资源能够以多种方式进行表述，包括将模型数据渲染为XML、JSON、Atom以及RSS的View实现；
- 可以使用ContentNegotiatingViewResolver来选择最适合客户端的表述；
- 借助@ResponseBody注解和各种HttpMethodConverter实现，能够替换基于视图的渲染方式；
- 类似地，@RequestBody注解以及HttpMethodConverter实现可以将传入的HTTP数据转化为传入控制器处理方法的Java对

象；

- 借助`RestTemplate`，Spring应用能够方便地使用`REST`资源。

本章中，我们将会介绍Spring RESTful的所有特性，首先介绍如何借助Spring MVC生成资源。然后在16.4小节中，我们会转向REST的客户端，看一下如何使用这些资源。那么，就从了解RESTful Spring MVC控制器是什么样子开始吧。

16.2 创建第一个REST端点

借助Spring的支持来实现REST功能有一个很有利的地方，那就是我们已经掌握了很多创建RESTful控制器的知识。从第5章到第7章中，我们学到了创建Web应用的知识，它们可以用在通过REST API暴露资源上。首先，我们会在名为`SpittleApiController`的新控制器中创建第一个REST端点。

如下的程序清单展现了这个新REST控制器起始的样子，它会提供`Spittle`资源。这是一个很简单的开始，但是在本章中，随着不断学习Spring REST编程模型的细节，我们将会不断构建这个控制器。

程序清单**16.1** 实现**RESTful**功能的**Spring MVC**控制器

```java
package spittr.api;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import spittr.Spittle;
import spittr.data.SpittleRepository;

@Controller
@RequestMapping("/spittles")
public class SpittleController {

  private static final String MAX_LONG_AS_STRING="9223372036854775807";

  private SpittleRepository spittleRepository;

  @Autowired
  public SpittleController(SpittleRepository spittleRepository) {
    this.spittleRepository = spittleRepository;
  }

  @RequestMapping(method=RequestMethod.GET)
  public List<Spittle> spittles(
      @RequestParam(value="max",
                    defaultValue=MAX_LONG_AS_STRING) long max,
      @RequestParam(value="count", defaultValue="20") int count) {

    return spittleRepository.findSpittles(max, count);
  }

}
```

让我们仔细看一下程序清单16.1。你能够看出来它服务于一个REST资源而不是Web页面吗？

可能看不出来！按照这个控制器的写法，并没有地方表明它是RESTful、服务于资源的控制器。实际上，你也许能够认出这个`spittles()`方法，我们曾经在第5章（5.3.1小节）见过它。

我们回忆一下，当发起对“`/spittles`”的`GET`请求时，将会调用`spittles()`方法。它会查找并返回`Spittle`列表，而这个列表会通过注入的`SpittleRepository`获取到。列表会放到模型中，用于视图的渲染。对于基于浏览器的Web应用，这可能意味着模型数据会渲染到HTML页面中。

但是，我们现在讨论的是创建REST API。在这种情况下，HTML并不是合适的数据表述形式。

表述是REST中很重要的一个方面。它是关于客户端和服务器端针对某一资源是如何通信的。任何给定的资源都几乎可以用任意的形式来进行表述。如果资源的使用者愿意使用JSON，那么资源就可以用JSON格式来表述。如果使用者喜欢尖括号，那相同的资源可以用XML来进行表述。同时，如果用户在浏览器中查看资源的话，可能更

愿意以HTML的方式来展现（或者PDF、Excel及其他便于人类阅读的格式）。资源没有变化——只是它的表述方式变化了。

> **注意：** 尽管Spring支持多种资源表述形式，但是在定义REST API的时候，不一定要全部使用它们。对于大多数客户端来说，用JSON和XML来进行表述就足够了。

当然，如果内容要由人类用户来使用的话，那么我们可能需要支持HTML格式的资源。根据资源的特点和应用的需求，我们还可能选择使用PDF文档或Excel表格来展现资源。

对于非人类用户的使用者，比如其他的应用或调用REST端点的代码，资源表述的首选应该是XML和JSON。借助Spring同时支持这两种方案非常简单，所以没有必要做一个非此即彼的选择。

按照我的意见，我推荐至少要支持JSON。JSON使用起来至少会像XML一样简单（很多人会说JSON会更加简单），并且如果客户端是JavaScript（最近一段时间以来，这种做法越来越常见）的话，JSON更是会成为优胜者，因为在JavaScript中使用JSON数据根本就不需要编排和解排（marshaling/demarshaling）。

需要了解的是控制器本身通常并不关心资源如何表述。控制器以Java对象的方式来处理资源。控制器完成了它的工作之后，资源才会被转化成最适合客户端的形式。

Spring提供了两种方法将资源的Java表述形式转换为发送给客户端的表述形式：

- 内容协商（*Content negotiation*）：选择一个视图，它能够将模型渲染为呈现给客户端的表述形式；
- 消息转换器（*Message conversion*）：通过一个消息转换器将控制器所返回的对象转换为呈现给客户端的表述形式。

鉴于我们在第5章和第6章中已经讨论过视图解析器，并且已经熟悉了基于视图的渲染（在第6章中），所以首先看一下如何使用内容协商来选择视图或视图解析器，它们将资源渲染为客户端能够接受的形式。

16.2.1 协商资源表述

你可以回忆一下在第5章中（以及图5.1所示），当控制器的处理方法完成时，通常会返回一个逻辑视图名。如果方法不直接返回逻辑视图

名（例如方法返回void），那么逻辑视图名会根据请求的URL判断得出。DispatcherServlet接下来会将视图的名字传递给一个视图解析器，要求它来帮助确定应该用哪个视图来渲染请求结果。

在面向人类访问的Web应用程序中，选择的视图通常来讲都会渲染为HTML。视图解析方案是个简单的一维活动。如果根据视图名匹配上了视图，那这就是我们要用的视图了。

当要将视图名解析为能够产生资源表述的视图时，我们就有另外一个维度需要考虑了。视图不仅要匹配视图名，而且所选择的视图要适合客户端。如果客户端想要JSON，那么渲染HTML的视图就不行了——尽管视图名可能匹配。

Spring的ContentNegotiatingViewResolver是一个特殊的视图解析器，它考虑到了客户端所需要的内容类型。按照其最简单的形式，ContentNegotiatingViewResolver可以按照下述形式进行配置：

```
@Bean
public ViewResolver cnViewResolver() {
  return new ContentNegotiatingViewResolver();
}
```

在这个简单的bean声明背后会涉及到很多事情。要理解ContentNegotiating-ViewResolver是如何工作的，这涉及内容协商的两个步骤：

1. 确定请求的媒体类型；

2. 找到适合请求媒体类型的最佳视图。

让我们深入了解每个步骤来了解ContentNegotiatingViewResolver是如何完成其任务的，首先从弄明白客户端需要什么类型的内容开始。

确定请求的媒体类型

在内容协商两步骤中，第一步是确定客户端想要什么类型的内容表述。表面上看，这似乎是一个很简单的事情。难道请求的Accept头部信息不是已经很清楚地表明要发送什么样的表述给客户端吗？

遗憾的是，Accept头部信息并不总是可靠的。如果客户端是Web浏览器，那并不能保证客户端需要的类型就是浏览器在Accept头部所

发送的值。Web浏览器一般只接受对人类用户友好的内容类型（如text/html），所以没有办法（除了面向开发人员的浏览器插件）指定不同的内容类型。

ContentNegotiatingViewResolver将会考虑到Accept头部信息并使用它所请求的媒体类型，但是它会首先查看URL的文件扩展名。如果URL在结尾处有文件扩展名的话，ContentNegotiatingViewResolver将会基于该扩展名确定所需的类型。如果扩展名是“.json”的话，那么所需的内容类型必须是“application/json”。如果扩展名是“.xml”，那么客户端请求的就是“application/xml”。当然，“.html”扩展名表明客户端所需的资源表述为HTML（text/html）。

如果根据文件扩展名不能得到任何媒体类型的话，那就会考虑请求中的Accept头部信息。在这种情况下，Accept头部信息中的值就表明了客户端想要的MIME类型，没有必要再去查找了。

最后，如果没有Accept头部信息，并且扩展名也无法提供帮助的话，ContentNegotiatingViewResolver将会使用“/”作为默认的内容类型，这就意味着客户端必须要接收服务器发送的任何形式的表述。

一旦内容类型确定之后，ContentNegotiatingViewResolver就该将逻辑视图名解析为渲染模型的View。与Spring的其他视图解析器不同，ContentNegotiatingViewResolver本身不会解析视图。而是委托给其他的视图解析器，让它们来解析视图。

ContentNegotiatingViewResolver要求其他的视图解析器将逻辑视图名解析为视图。解析得到的每个视图都会放到一个列表中。这个列表装配完成后，ContentNegotiatingViewResolver会循环客户端请求的所有媒体类型，在候选的视图中查找能够产生对应内容类型的视图。第一个匹配的视图会用来渲染模型。

影响媒体类型的选择

在上述的选择过程中，我们阐述了确定所请求媒体类型的默认策略。但是通过为其设置一个ContentNegotiationManager，我们能够改变它的行为。借助Content-NegotiationManager我们所能做到的事情如下所示：

- 指定默认的内容类型，如果根据请求无法得到内容类型的话，将

会使用默认值；

- 通过请求参数指定内容类型；
- 忽视请求的Accept头部信息；
- 将请求的扩展名映射为特定的媒体类型；
- 将JAF（Java Activation Framework）作为根据扩展名查找媒体类型的备用方案。

有三种配置ContentNegotiationManager的方法：

- 直接声明一个ContentNegotiationManager类型的bean；
- 通过ContentNegotiationManagerFactoryBean间接创建bean；
- 重载WebMvcConfigurerAdapter的configureContentNegotiation()方法。

直接创建ContentNegotiationManager有一些复杂，除非有充分的原因，否则我们不会愿意这样做。后两种方案能够让创建ContentNegotiationManager更加简单。

ContentNegotiationManager是在Spring 3.2中加入的

ContentNegotiationManager是Spring中相对比较新的功能，是在Spring 3.2中引入的。在Spring 3.2之前，ContentNegotiatingViewResolver的很多行为都是通过直接设置ContentNegotiatingViewResolver的属性进行配置的。从Spring 3.2开始，Content-NegotiatingViewResolver的大多数Setter方法都废弃了，鼓励通过Content-NegotiationManager来进行配置。

尽管我不会在本章中介绍配置ContentNegotiatingViewResolver的旧方法，但是我们在创建ContentNegotiationManager所设置的很多属性，在ContentNegotiatingViewResolver中都有对应的属性。如果你使用较早版本的Spring的话，应该能够很容易地将新的配置方式对应到旧配置方式中。

一般而言，如果我们使用XML配置ContentNegotiationManager的话，那最有用的将会是ContentNegotiationManagerFactoryBean。例如，我们可能希望在XML中配置ContentNegotiationManager使用“application/json”作为默认的内容类型：

```
<bean id="contentNegotiationManager"
  class="org.springframework.http.ContentNegotiationManagerFactoryBean"
  p:defaultContentType="application/json">
```

因为ContentNegotiationManagerFactoryBean是FactoryBean的实现，所以它会创建一个ContentNegotiationManager bean。这个ContentNegotiationManager能够注入到ContentNegotiatingViewResolver的contentNegotiationManager属性中。

如果使用Java配置的话，获得ContentNegotiationManager的最简便方法就是扩展WebMvcConfigurerAdapter并重载configureContentNegotiation()方法。在创建Spring MVC应用的时候，我们很可能已经扩展了WebMvcConfigurerAdapter。例如，在Spittr应用中，我们已经有了WebMvcConfigurerAdapter的扩展类，名为WebConfig，所以需要做的就是重载configureContentNegotiation()方法。如下就是configureContentNegotiation()的一个实现，它设置了默认的内容类型：

```
@Override
public void configureContentNegotiation(
    ContentNegotiationConfigurer configurer) {
  configurer.defaultContentType(MediaType.APPLICATION_JSON);
}
```

我们可以看到，configureContentNegotiation()方法给定了一个Content-NegotiationConfigurer对象。ContentNegotiationConfigurer中的一些方法对应于ContentNegotiationManager的Setter方法，这样我们就能在ContentNegotiation-Manager创建时，设置任意内容协商相关的属性。在本例中，我们调用defaultContentType()方法将默认的内容类型设置为“application/json”。

现在，我们已经有了ContentNegotiationManagerbean，接下来就需要将它注入到ContentNegotiatingViewResolver的contentNegotiationManager属性中。这需要我们稍微修改一下之前声明ContentNegotiatingViewResolver的@Bean方法：

```
@Bean
public ViewResolver cnViewResolver(ContentNegotiationManager cnm) {
  ContentNegotiatingViewResolver cnvr =
      new ContentNegotiatingViewResolver();
  cnvr.setContentNegotiationManager(cnm);
  return cnvr;
}
```

这个@Bean方法注入了ContentNegotiationManager，并使用它调用了setContentNegotiationManager()。这样的结果就是ContentNegotiatingView、Resolver将会使用ContentNegotiationManager所定义的行为。

配置ContentNegotiationManager有很多的细节，在这里无法对它们进行一一介绍。如下的程序清单是一个非常简单的配置样例，当我使用ContentNegotiating-ViewResolver的时候，通常会采用这种用法：它默认会使用HTML视图，但是对特定的视图名称将会渲染为JSON输出。

程序清单**16.2**　配置**ContentNegotiationManager**

```
@Bean
public ViewResolver cnViewResolver(ContentNegotiationManager cnm) {
  ContentNegotiatingViewResolver cnvr =
      new ContentNegotiatingViewResolver();
  cnvr.setContentNegotiationManager(cnm);
  return cnvr;
}
@Override
public void configureContentNegotiation(
    ContentNegotiationConfigurer configurer) {
  configurer.defaultContentType(MediaType.TEXT_HTML);      <— 默认为 HTML
}

@Bean
public ViewResolver beanNameViewResolver() {               <— 以 bean 的形式查找视图
  return new BeanNameViewResolver();
}

@Bean
public View spittles() {
  return new MappingJackson2JsonView();                    <— 将“spittles”定义为 JSON 视图
}
```

除了程序清单16.2中的内容以外，还应该有一个能够处理HTML的视图解析器（如InternalResourceViewResolver或TilesViewResolver）。在大多数场景下，ContentNegotiatingViewResolver会假设客户端需要HTML，如ContentNegotiationManager配置所示。但是，如果客户端指定了它想要JSON（通过在请求路径上使用“.json”扩展名或Accept头部信息）的话，那么ContentNegotiatingViewResolver将会查找能够处理JSON视图的视图解析器。

如果逻辑视图的名称为“spittles”，那么我们所配置的BeanNameViewResolver将会解析spittles()方法中所声明的View。这是因为bean名称匹配逻辑视图的名称。如果没有匹配的View的话，ContentNegotiatingViewResolver将会采用默认的行

为，将其输出为HTML。

ContentNegotiatingViewResolver一旦能够确定客户端想要什么样的媒体类型，接下来就是查找渲染这种内容的视图。

ContentNegotiatingViewResolver的优势与限制

ContentNegotiatingViewResolver最大的优势在于，它在Spring MVC之上构建了REST资源表述层，控制器代码无需修改。相同的一套控制器方法能够为面向人类的用户产生HTML内容，也能针对不是人类的客户端产生JSON或XML。

如果面向人类用户的接口与面向非人类客户端的接口之间有很多重叠的话，那么内容协商是一种很便利的方案。在实践中，面向人类用户的视图与REST API在细节上很少能够处于相同的级别。如果面向人类用户的接口与面向非人类客户端的接口之间没有太多重叠的话，那么ContentNegotiatingViewResolver的优势就体现不出来了。

ContentNegotiatingViewResolver还有一个严重的限制。作为ViewResolver的实现，它只能决定资源该如何渲染到客户端，并没有涉及到客户端要发送什么样的表述给控制器使用。如果客户端发送JSON或XML的话，那么ContentNegotiatingViewResolver就无法提供帮助了。

ContentNegotiatingViewResolver还有一个相关的小问题，所选中的View会渲染模型给客户端，而不是资源。这里有个细微但很重要的区别。当客户端请求JSON格式的Spittle对象列表时，客户端希望得到的响应可能如下所示：

```
[
  {
    "id": 42,
    "latitude": 28.419489,
    "longitude": -81.581184,
    "message": "Hello World!",
    "time": 1400389200000
  },
  {
    "id": 43,
    "latitude": 28.419136,
    "longitude": -81.577225,
    "message": "Blast off!",
    "time": 1400475600000
  }
]
```

而模型是key-value组成的Map，那么响应可能会如下所示：

```
{
    "spittleList": [
        {
            "id": 42,
            "latitude": 28.419489,
            "longitude": -81.581184,
            "message": "Hello World!",
            "time": 1400389200000
        },
        {
            "id": 43,
            "latitude": 28.419136,
            "longitude": -81.577225,
            "message": "Blast off!",
            "time": 1400475600000
        }
    ]
}
```

尽管这不是很严重的问题，但确实可能不是客户端所预期的结果。

因为有这些限制，我通常建议不要使用ContentNegotiatingViewResolver。我更加倾向于使用Spring的消息转换功能来生成资源表述。接下来，我们看一下如何在控制器代码中使用Spring的消息转换器。

16.2.2 使用HTTP信息转换器

消息转换（message conversion）提供了一种更为直接的方式，它能够将控制器产生的数据转换为服务于客户端的表述形式。当使用消息转换功能时，DispatcherServlet不再需要那么麻烦地将模型数据传送到视图中。实际上，这里根本就没有模型，也没有视图，只有控制器产生的数据，以及消息转换器（message converter）转换数据之后所产生的资源表述。

Spring自带了各种各样的转换器，如表16.1所示，这些转换器满足了最常见的将对象转换为表述的需要。

例如，假设客户端通过请求的Accept头信息表明它能接受“application/json”，并且Jackson JSON在类路径下，那么处理方法返回的对象将交给MappingJacksonHttp-MessageConverter，并由它转换为返回客户端的JSON表述形式。另一方面，如果请求的头信息表明客户端想要“text/xml”格式，那么Jaxb2RootElementHttpMessage-Converter将会为客户端产生XML响应。

注意，表16.2中的HTTP信息转换器除了其中的五个以外都是自动注册的，所以要使用它们的话，不需要Spring配置。但是为了支持它

们，你需要添加一些库到应用程序的类路径下。例如，如果你想使用MappingJacksonHttpMessageConverter来实现JSON消息和Java对象的互相转换，那么需要将Jackson JSON Processor库添加到类路径中。类似地，如果你想使用Jaxb2RootElementHttpMessageConverter来实现XML消息和Java对象的互相转换，那么需要JAXB库。如果信息是Atom或RSS格式的话，那么Atom-FeedHttpMessageConverter和RssChannelHttpMessageConverter会需要Rome库。

表**16.1　Spring**提供了多个**HTTP**信息转换器，用于实现资源表述与各种**Java**类型之间的互相转换

信息转换器	描　　述
AtomFeedHttpMessageConverter	Rome Feed对象和Atom feed（媒体类型application/atom+xml）之间的互相转换。 如果 *Rome* 包在类路径下将会进行注册
BufferedImageHttpMessageConverter	BufferedImages与图片二进制数据之间互相转换
ByteArrayHttpMessageConverter	读取/写入字节数组。从所有媒体类型（*/*）中读取，并以application/octet-stream格式写入
FormHttpMessageConverter	将application/x-www-form-urlencoded内容读入到MultiValueMap<String,String>中，也会将MultiValueMap<String,String>写入到application/x-www-form- urlencoded中或将MultiValueMap<String, Object>写入到multipart/form-data中
Jaxb2RootElementHttpMessageConverter	在XML（text/xml或application/xml）和使用JAXB2注解的对象间互相读取和写入。 如果 *JAXB v2* 库在类路径下，将进行注册
	在JSON和类型化的对象或非类型化的

MappingJacksonHttpMessageConverter	HashMap间互相读取和写入。 如果 *Jackson JSON* 库在类路径下，将进行注册
MappingJackson2HttpMessageConverter	在JSON和类型化的对象或非类型化的HashMap间互相读取和写入。 如果 *Jackson 2 JSON* 库在类路径下，将进行注册
MarshallingHttpMessageConverter	使用注入的编排器和解排器（marshaller和unmarshaller）来读入和写入XML。支持的编排器和解排器包括Castor、JAXB2、JIBX、XMLBeans以及Xstream
ResourceHttpMessageConverter	读取或写入Resource
RssChannelHttpMessageConverter	在RSS feed和Rome Channel对象间互相读取或写入。 如果 *Rome* 库在类路径下，将进行注册
SourceHttpMessageConverter	在XML和javax.xml.transform.Source对象间互相读取和写入。 默认注册
StringHttpMessageConverter	将所有媒体类型（*/*）读取为String。将String写入为text/plain
XmlAwareFormHttpMessageConverter	FormHttpMessageConverter的扩展，使用SourceHttp MessageConverter来支持基于XML的部分

你可能已经猜到了，为了支持消息转换，我们需要对Spring MVC的编程模型进行一些小调整。

在响应体中返回资源状态

正常情况下，当处理方法返回Java对象（除String外或View的实现以外）时，这个对象会放在模型中并在视图中渲染使用。但是，如果使用了消息转换功能的话，我们需要告诉Spring跳过正常的模型/视图流

程，并使用消息转换器。有不少方式都能做到这一点，但是最简单的方法是为控制器方法添加@ResponseBody注解。

重新看一下程序清单16.1中的spittles()方法，我们可以为其添加@ResponseBody注解，这样就能让Spring将方法返回的List<Spittle>转换为响应体：

```
@RequestMapping(method=RequestMethod.GET,
                produces="application/json")
public @ResponseBody List<Spittle> spittles(
    @RequestParam(value="max",
                  defaultValue=MAX_LONG_AS_STRING) long max,
    @RequestParam(value="count", defaultValue="20") int count) {

  return spittleRepository.findSpittles(max, count);
}
```

@ResponseBody注解会告知Spring，我们要将返回的对象作为资源发送给客户端，并将其转换为客户端可接受的表述形式。更具体地讲，DispatcherServlet将会考虑到请求中Accept头部信息，并查找能够为客户端提供所需表述形式的消息转换器。

举例来讲，假设客户端的Accept头部信息表明它接受“application/json”，并且Jackson JSON库位于应用的类路径下，那么将会选择MappingJacksonHttpMessage-Converter或MappingJackson2HttpMessageConverter（这取决于类路径下是哪个版本的Jackson）。消息转换器会将控制器返回的Spittle列表转换为JSON文档，并将其写入到响应体中。响应大致会如下所示：

```
[
  {
    "id": 42,
    "latitude": 28.419489,
    "longitude": -81.581184,
    "message": "Hello World!",
    "time": 1400389200000
  },
  {
    "id": 43,
    "latitude": 28.419136,
    "longitude": -81.577225,
    "message": "Blast off!",
    "time": 1400475600000
  }
]
```

Jackson默认会使用反射

注意在默认情况下，Jackson JSON库在将返回的对象转换为JSON资源表述时，会使用反射。对于简单的表述内容来讲，这没有什么问题。但是如果你重构了Java类型，比如添加、移除或重命名属性，那么所

产生的JSON也将会发生变化（如果客户端依赖这些属性的话，那客户端有可能会出错）。

但是，我们可以在Java类型上使用Jackson的映射注解，从而改变产生JSON的行为。这样我们就能更多地控制所产生的JSON，从而防止它影响到API或客户端。

Jackson映射注解的内容超出了本书的讨论范围，不过关于这个主题，在http://wiki.fasterxml.com/Jackson-Annotations上有一些有用的文档。

谈及Accept头部信息，请注意getSpitter()的@RequestMapping注解。在这里，我使用了produces属性表明这个方法只处理预期输出为JSON的请求。也就是说，这个方法只会处理Accept头部信息包含“application/json”的请求。其他任何类型的请求，即使它的URL匹配指定的路径并且是GET请求也不会被这个方法处理。这样的请求会被其他的方法来进行处理（如果存在适当方法的话），或者返回客户端HTTP 406（Not Acceptable）响应。

在请求体中接收资源状态

到目前为止，我们只关注了REST端点如何为客户端提供资源。但是REST并不是只读的，REST API也可以接受来自客户端的资源表述。如果要让控制器将客户端发送的JSON和XML转换为它所使用的Java对象，那是非常不方便的。在处理逻辑离开控制器的时候，Spring的消息转换器能够将对象转换为表述——它们能不能在表述传入的时候完成相同的任务呢？

@ResponseBody能够告诉Spring在把数据发送给客户端的时候，要使用某一个消息器，与之类似，@RequestBody也能告诉Spring查找一个消息转换器，将来自客户端的资源表述转换为对象。例如，假设我们需要一种方式将客户端提交的新Spittle保存起来。我们可以按照如下的方式编写控制器方法来处理这种请求：

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
public @ResponseBody
    Spittle saveSpittle(@RequestBody Spittle spittle) {
  return spittleRepository.save(spittle);
}
```

如果忽略掉注解的话，那saveSpittle()是一个非常简单的方法。它接受一个Spittle对象作为参数，并使用SpittleRepository进行保存，最终返回spittleRepository.save()方法所得到的

Spittle对象。

但是，通过使用注解，它会变得更加有意思也更加强大。@RequestMapping表明它只能处理“/spittles”（在类级别的@RequestMapping中进行了声明）的POST请求。POST请求体中预期要包含一个Spittle的资源表述。因为Spittle参数上使用了@RequestBody，所以Spring将会查看请求中的Content-Type头部信息，并查找能够将请求体转换为Spittle的消息转换器。

例如，如果客户端发送的Spittle数据是JSON表述形式，那么Content-Type头部信息可能就会是“application/json”。在这种情况下，DispatcherServlet会查找能够将JSON转换为Java对象的消息转换器。如果Jackson 2库在类路径中，那么MappingJackson2HttpMessageConverter将会担此重任，将JSON表述转换为Spittle，然后传递到saveSpittle()方法中。这个方法还使用了@ResponseBody注解，因此方法返回的Spittle对象将会转换为某种资源表述，发送给客户端。

注意，@RequestMapping有一个consumes属性，我们将其设置为“application/ json”。consumes属性的工作方式类似于produces，不过它会关注请求的Content-Type头部信息。它会告诉Spring这个方法只会处理对“/spittles”的POST请求，并且要求请求的Content-Type头部信息为“application/json”。如果无法满足这些条件的话，会由其他方法（如果存在合适的方法的话）来处理请求。

为控制器默认设置消息转换

当处理请求时，@ResponseBody和@RequestBody是启用消息转换的一种简洁和强大方式。但是，如果你所编写的控制器有多个方法，并且每个方法都需要信息转换功能的话，那么这些注解就会带来一定程度的重复性。

Spring 4.0引入了@RestController注解，能够在这个方面给我们提供帮助。如果在控制器类上使用@RestController来代替@Controller的话，Spring将会为该控制器的所有处理方法应用消息转换功能。我们不必为每个方法都添加@ResponseBody了。我们所定义的SpittleController可能就会如下所示：

程序清单**16.3**　使用**@RestController**注解

```
package spittr.api;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import spittr.Spittle;
import spittr.data.SpittleRepository;

@RestController                                  <—默认使用消息转换
@RequestMapping("/spittles")
public class SpittleController {
  private static final String MAX_LONG_AS_STRING="9223372036854775807";

  private SpittleRepository spittleRepository;

  @Autowired
  public SpittleController(SpittleRepository spittleRepository) {
    this.spittleRepository = spittleRepository;
  }

  @RequestMapping(method=RequestMethod.GET)
  public List<Spittle> spittles(
      @RequestParam(value="max",
                    defaultValue=MAX_LONG_AS_STRING) long max,
      @RequestParam(value="count", defaultValue="20") int count) {

    return spittleRepository.findSpittles(max, count);
  }

  @RequestMapping(
      method=RequestMethod.POST
      consumes="application/json")
  public Spittle saveSpittle(@RequestBody Spittle spittle) {
    return spittleRepository.save(spittle);
  }
}
```

程序清单16.3的关键点在于代码中此时不包含什么。这两个处理器方法都没有使用`@ResponseBody`注解，因为控制器使用了`@RestController`，所以它的方法所返回的对象将会通过消息转换机制，产生客户端所需的资源表述。

到目前为止，我们看到了如何使用Spring MVC编程模型将RESTful资源发布到响应体之中。但是响应除了负载以外还会有其他的内容。头部信息和状态码也能够为客户端提供响应的有用信息。接下来，我们看一下在提供资源的时候，如何填充头部信息和设置状态码。

16.3 提供资源之外的其他内容

`@ResponseBody`提供了一种很有用的方式，能够将控制器返回的Java对象转换为发送到客户端的资源表述。实际上，将资源表述发送给客户端只是整个过程的一部分。一个好的REST API不仅能够在客户端和服务器之间传递资源，它还能够给客户端提供额外的元数据，帮助客户端理解资源或者在请求中出现了什么情况。

16.3.1　发送错误信息到客户端

例如，我们为SpittleController添加一个新的处理器方法，它会提供单个Spittle对象：

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public @ResponseBody Spittle spittleById(@PathVariable long id) {
  return spittleRepository.findOne(id);
}
```

在这里，通过id参数传入了一个ID，然后根据它调用Repository的findOne()方法，查找Spittle对象。处理器方法会返回findOne()方法得到的Spittle对象，消息转换器会负责产生客户端所需的资源表述。

非常简单，对吧？我们没办法让它更棒了。它还能更好吗？

如果根据给定的ID，无法找到某个Spittle对象的ID属性能够与之匹配，findOne()方法返回null的时候，你觉得会发生什么呢？

结果就是spittleById()方法会返回null，响应体为空，不会返回任何有用的数据给客户端。同时，响应中默认的HTTP状态码是200（OK），表示所有的事情运行正常。

但是，所有的事情都是不对的。客户端要求Spittle对象，但是它什么都没有得到。它既没有收到Spittle对象也没有收到任何消息表明出现了错误。服务器实际上是在说："这是一个没用的响应，但是能够告诉你一切都正常！"

现在，我们考虑一下在这种场景下应该发生什么。至少，状态码不应该是200，而应该是404（Not Found），告诉客户端它们所要求的内容没有找到。如果响应体中能够包含错误信息而不是空的话就更好了。

Spring提供了多种方式来处理这样的场景：

- 使用@ResponseStatus注解可以指定状态码；
- 控制器方法可以返回ResponseEntity对象，该对象能够包含更多响应相关的元数据；
- 异常处理器能够应对错误场景，这样处理器方法就能关注于正常的状况。

在这个方面，Spring提供了很多的灵活性，其实也不存在唯一正确的

方式。我不会用某一种固定的策略来处理所有的错误或涵盖所有的场景，而是会向读者展现多种修改spittleById()的方法，以应对Spittle无法找到的场景。

使用ResponseEntity

作为@ResponseBody的替代方案，控制器方法可以返回一个ResponseEntity对象。ResponseEntity中可以包含响应相关的元数据（如头部信息和状态码）以及要转换成资源表述的对象。

因为ResponseEntity允许我们指定响应的状态码，所以当无法找到Spittle的时候，我们可以返回HTTP 404错误。如下是新版本的spittleById()，它会返回ResponseEntity：

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  HttpStatus status = spittle != null ?
                      HttpStatus.OK : HttpStatus.NOT_FOUND;
  return new ResponseEntity<Spittle>(spittle, status);
}
```

像前面一样，路径中得到的ID用来从Repository中检索Spittle。如果找到的话，状态码设置为HttpStatus.OK（这是之前的默认值），但是如果Repository返回null的话，状态码设置为HttpStatus.NOT_FOUND，这会转换为HTTP 404。最后，会创建一个新的ResponseEntity，它会把Spittle和状态码传送给客户端。

注意这个spittleById()方法没有使用@ResponseBody注解。除了包含响应头信息、状态码以及负载以外，ResponseEntity还包含了@ResponseBody的语义，因此负载部分将会渲染到响应体中，就像之前在方法上使用@ResponseBody注解一样。如果返回ResponseEntity的话，那就没有必要在方法上使用@ResponseBody注解了。

我们在正确的方向上走出了第一步，如果所要求的Spittle无法找到的话，客户端能够得到一个合适的状态码。但是在本例中，响应体依然为空。我们可能会希望在响应体中包含一些错误信息。

我们重试一次，首先定义一个包含错误信息的Error对象：

```
public class Error {
  private int code;
  private String message;

  public Error(int code, String message) {
    this.code = code;
    this.message = message;
  }

  public int getCode() {
    return code;
  }

  public String getMessage() {
    return message;
  }
}
```

然后，我们可以修改spittleById()，让它返回Error：

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<?> spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  if (spittle == null) {
    Error error = new Error(4, "Spittle [" + id + "] not found");
    return new ResponseEntity<Error>(error, HttpStatus.NOT_FOUND);
  }
  return new ResponseEntity<Spittle>(spittle, HttpStatus.OK);
}
```

现在，这个方法的行为已经符合我们的预期了。如果找到Spittle的话，就会把返回的对象以及200（OK）的状态码封装到ResponseEntity中。另一方面，如果findOne()返回null的话，将会创建一个Error对象，并将其与404（Not Found）状态码一起封装到ResponseEntity中，然后返回。

你也许觉得我们可以到此结束这个话题了。毕竟，方法按照我们期望的方式在运行。但是，还有一点事情让我不太舒服。

首先，这比我们开始的时候更为复杂。涉及到了更多的逻辑，包括条件语句。另外，方法返回ResponseEntity<?>感觉有些问题。ResponseEntity所使用的泛型为它的解析或出现错误留下了太多的空间。

不过，我们可以借助错误处理器来修正这些问题。

处理错误

spittleById()方法中的if代码块是处理错误的，但这是控制器中错误处理器（error handler）所擅长的领域。错误处理器能够处理导致问题的场景，这样常规的处理器方法就能只关心正常的逻辑处理路径了。

我们重构一下代码来使用错误处理器。首先，定义能够对应SpittleNotFound-Exception的错误处理器：

```
@ExceptionHandler(SpittleNotFoundException.class)
public ResponseEntity<Error> spittleNotFound(
                                    SpittleNotFoundException e) {
  long spittleId = e.getSpittleId();
  Error error = new Error(4, "Spittle [" + spittleId + "] not found");
  return new ResponseEntity<Error>(error, HttpStatus.NOT_FOUND);
}
```

@ExceptionHandler注解能够用到控制器方法中，用来处理特定的异常。这里，它表明如果在控制器的任意处理方法中抛出SpittleNotFoundException异常，就会调用spittleNotFound()方法来处理异常。

至于SpittleNotFoundException，它是一个很简单异常类：

```
public class SpittleNotFoundException extends RuntimeException {
  private long spittleId;
  public SpittleNotFoundException(long spittleId) {
    this.spittleId = spittleId;
  }

  public long getSpittleId() {
    return spittleId;
  }
}
```

现在，我们可以移除掉spittleById()方法中大多数的错误处理代码：

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  if (spittle == null) { throw new SpittleNotFoundException(id); }
  return new ResponseEntity<Spittle>(spittle, HttpStatus.OK);
}
```

这个版本的spittleById()方法确实干净了很多。除了对返回值进行null检查，它完全关注于成功的场景，也就是能够找到请求的Spittle。同时，在返回类型中，我们能移除掉奇怪的泛型了。

不过，我们能够让代码更加干净一些。现在我们已经知道spittleById()将会返回Spittle并且HTTP状态码始终会是200（OK），那么就可以不再使用ResponseEntity，而是将其替换为@ResponseBody：

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public @ResponseBody Spittle spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  if (spittle == null) { throw new SpittleNotFoundException(id); }
  return spittle;
}
```

当然，如果控制器类上使用了@RestController，我们甚至不再需要@ResponseBody：

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public Spittle spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  if (spittle == null) { throw new SpittleNotFoundException(id); }
  return spittle;
}
```

鉴于错误处理器的方法会始终返回Error，并且HTTP状态码为404（Not Found），那么现在我们可以对spittleNotFound()方法进行类似的清理：

```
@ExceptionHandler(SpittleNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public @ResponseBody Error spittleNotFound(SpittleNotFoundException e) {
  long spittleId = e.getSpittleId();
  return new Error(4, "Spittle [" + spittleId + "] not found");
}
```

因为spittleNotFound()方法始终会返回Error，所以使用ResponseEntity的唯一原因就是能够设置状态码。但是通过为spittleNotFound()方法添加@ResponseStatus(HttpStatus.NOT_FOUND)注解，我们可以达到相同的效果，而且可以不再使用ResponseEntity了。

同样，如果控制器类上使用了@RestController，那么就可以移除掉@ResponseBody，让代码更加干净：

```
@ExceptionHandler(SpittleNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public Error spittleNotFound(SpittleNotFoundException e) {
  long spittleId = e.getSpittleId();
  return new Error(4, "Spittle [" + spittleId + "] not found");
}
```

在一定程度上，我们已经圆满达到了想要的效果。为了设置响应状态码，我们首先使用ResponseEntity，但是稍后我们借助异常处理器以及@ResponseStatus，避免使用ResponseEntity，从而让代码更加整洁。

似乎，我们不再需要使用ResponseEntity了。但是，有一种场

景ResponseEntity能够很好地完成，但是其他的注解或异常处理器却做不到。现在，我们看一下如何在响应中设置头部信息。

16.3.2 在响应中设置头部信息

在saveSpittle()方法中，我们在处理POST请求的过程中创建了一个新的Spittle资源。但是，按照目前的写法（参考程序清单16.3），我们无法准确地与客户端交流。

在saveSpittle()处理完请求之后，服务器在响应体中包含了Spittle的表述以及HTTP状态码200（OK），将其返回给客户端。这里没有什么大问题，但是还不是完全准确。

当然，假设处理请求的过程中成功创建了资源，状态可以视为OK。但是，我们不仅仅需要说“OK”。我们创建了新的内容，HTTP状态码也将这种情况告诉给了客户端。不过，HTTP 201不仅能够表明请求成功完成，而且还能描述创建了新资源。如果我们希望完整准确地与客户端交流，那么响应是不是应该为201（Created），而不仅仅是200（OK）呢？

根据我们目前所学到的知识，这个问题解决起来很容易。我们需要做的就是为saveSpittle()方法添加@ResponseStatus注解，如下所示：

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Spittle saveSpittle(@RequestBody Spittle spittle) {
  return spittleRepository.save(spittle);
}
```

这应该能够完成我们的任务，现在状态码能够精确反应发生了什么情况。它告诉客户端我们新创建了资源。问题已经得以解决！

但这只是问题的一部分。客户端知道新创建了资源，你觉得客户端会不会感兴趣新创建的资源在哪里呢？毕竟，这是一个新创建的资源，会有一个新的URL与之关联。难道客户端只能猜测新创建资源的URL是什么吗？我们能不能以某种方式将其告诉客户端？

当创建新资源的时候，将资源的URL放在响应的Location头部信息中，并返回给客户端是一种很好的方式。因此，我们需要有一种方式来填充响应头部信息，此时我们的老朋友ResponseEntity就能提

供帮助了。

如下的程序清单展现了一个新版本的saveSpittle()，它会返回ResponseEntity用来告诉客户端新创建的资源。

程序清单**16.4**　当返回**ResponseEntity**时，在响应中设置头部信息

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
public ResponseEntity<Spittle> saveSpittle(
                      @RequestBody Spittle spittle) {

  Spittle spittle = spittleRepository.save(spittle);      <— 获取 Spittle

  HttpHeaders headers = new HttpHeaders();       <— 设置 Location 头部信息
   URI locationUri = URI.create(
      "http://localhost:8080/spittr/spittles/" + spittle.getId());
  headers.setLocation(locationUri);

  ResponseEntity<Spittle> responseEntity =       <— 创建 ResponseEntity
       new ResponseEntity<Spittle>(
          spittle, headers, HttpStatus.CREATED)
  return responseEntity;
}
```

在这个新的版本中，我们创建了一个HttpHeaders实例，用来存放希望在响应中包含的头部信息值。HttpHeaders是MultiValueMap<String, String>的特殊实现，它有一些便利的Setter方法（如setLocation()），用来设置常见的HTTP头部信息。在得到新创建Spittle资源的URL之后，接下来使用这个头部信息来创建ResponseEntity。

哇！原本简单的saveSpittle()方法瞬间变得臃肿了。但是，更值得关注的是，它使用硬编码值的方式来构建Location头部信息。URL中“localhost”以及“8080”这两个部分尤其需要注意，因为如果我们将应用部署到其他地方，而不是在本地运行的话，它们就不适用了。

我们其实没有必要手动构建URL，Spring提供了UriComponentsBuilder，可以给我们一些帮助。它是一个构建类，通过逐步指定URL中的各种组成部分（如host、端口、路径以及查询），我们能够使用它来构建UriComponents实例。借助UriComponentsBuilder所构建的UriComponents对象，我们就能获得适合设置给Location头部信息的URI。

为了使用UriComponentsBuilder，我们需要做的就是在处理器方法中将其作为一个参数，如下面的程序清单所示。

程序清单**16.5**　使用**UriComponentsBuilder**来构建**Location URI**

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
public ResponseEntity<Spittle> saveSpittle(
      @RequestBody Spittle spittle,
      UriComponentsBuilder ucb) {          <— 给定 UriComponentsBuilder ...

  Spittle spittle = spittleRepository.save(spittle);

  HttpHeaders headers = new HttpHeaders();     <— ... 计算 Location URI
   URI locationUri =
      ucb.path("/spittles/")
         .path(String.valueOf(spittle.getId()))
         .build()
         .toUri();
  headers.setLocation(locationUri);

  ResponseEntity<Spittle> responseEntity =
      new ResponseEntity<Spittle>(
          spittle, headers, HttpStatus.CREATED)
  return responseEntity;
}
```

在处理器方法所得到的UriComponentsBuilder中，会预先配置已知的信息如host、端口以及Servlet内容。它会从处理器方法所对应的请求中获取这些基础信息。基于这些信息，代码会通过设置路径的方式构建UriComponents其余的部分。

注意，路径的构建分为两步。第一步调用path()方法，将其设置为“/ spittles/”，也就是这个控制器所能处理的基础路径。然后，在第二次调用path()的时候，使用了已保存Spittle的ID。我们可以推断出来，每次调用path()都会基于上次调用的结果。

在路径设置完成之后，调用build()方法来构建UriComponents对象，根据这个对象调用toUri()就能得到新创建Spittle的URI。

在REST API中暴露资源只代表了会话的一端。如果发布的API没有人关心和使用的话，那也没有什么价值。通常来讲，移动或JavaScript应用会是REST API的客户端，但是Spring应用也完全可以使用这些资源。我们换个方向，看一下如何编写Spring代码实现RESTful交互的客户端。

16.4　编写REST客户端

作为客户端，编写与REST资源交互的代码可能会比较乏味，并且所编写的代码都是样板式的。例如，假设我们需要借助Facebook的Graph API，编写方法来获取某人的Facebook基本信息。不过，获取基

本信息的代码会有点复杂，如下面的程序清单所示。

程序清单**16.6** 使用**Apache HTTP Client**获取**Facebook**中的个人基本信息

```
public Profile fetchFacebookProfile(String id) {
  try {
    HttpClient client = HttpClients.createDefault();
    HttpGet request = new HttpGet("http://graph.facebook.com/" + id);
     request.setHeader("Accept", "application/json");

    HttpResponse response = client.execute(request);

    HttpEntity entity = response.getEntity();
    ObjectMapper mapper = new ObjectMapper();
         return mapper.readValue(entity.getContent(), Profile.class);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
```

创建请求

创建客户端

将响应映射为对象

执行请求

你可以看到，在使用REST资源的时候涉及很多代码。这里我甚至还偷懒使用了Jakarta Commons HTTP Client发起请求并使用Jackson JSON processor解析响应。

仔细看一下`fetchFacebookProfile()`方法，你可能会发现方法中只有少量代码与获取Facebook个人信息直接相关。如果你要编写另一个方法来使用其他的REST资源，很可能会有很多代码是与`fetchFacebookProfile()`相同的。

另外，还有一些地方可能会抛出的`IOException`异常。因为`IOException`是检查型异常，所以要么捕获它，要么抛出它。在本示例中，我选择捕获它并在它的位置重新抛出一个非检查型异常`RuntimeException`。

鉴于在资源使用上有如此之多的样板代码，你可能会觉得最好的方式是封装通用代码并参数化可变的部分。这正是Spring的`RestTemplate`所做的事情。就像`JdbcTemplate`处理了JDBC数据访问时的丑陋部分，`RestTemplate`让我们在使用RESTful资源时免于编写那些乏味的代码。

稍后，我们将会看到如何借助`RestTemplate`重写`fetchFacebookProfile()`方法，这会戏剧性的简化该方法并消除掉样板式代码。但首先，让我们整体了解一下`RestTemplate`提供的所有REST操作。

16.4.1 了解**RestTemplate**的操作

RestTemplate定义了36个与REST资源交互的方法，其中的大多数都对应于HTTP的方法。但是，在本章中我没有足够的篇幅涵盖所有的36个方法。其实，这里面只有11个独立的方法，其中有十个有三种重载形式，而第十一个则重载了六次，这样一共形成了36个方法。表16.2描述了RestTemplate所提供的11个独立方法。

除了TRACE以外，RestTemplate涵盖了所有的HTTP动作。除此之外，execute()和exchange()提供了较低层次的通用方法来使用任意的HTTP方法。

表16.2中的大多数操作都以三种方法的形式进行了重载：

- 一个使用java.net.URI作为URL格式，不支持参数化URL；
- 一个使用String作为URL格式，并使用Map指明URL参数；
- 一个使用String作为URL格式，并使用可变参数列表指明URL参数。

明确了RestTemplate所提供的11个操作以及各个变种如何工作之后，你就能以自己的方式编写使用REST资源的客户端了。我们通过对四个主要HTTP方法的支持（也就是GET、PUT、DELETE和POST）来研究RestTemplate的操作。我们从GET方法的getForObject()和getForEntity()开始。

表**16.2　RestTemplate**定义了**11**个独立的操作，而每一个都有重载，这样一共是**36**个方法

方　　法	描　　述
delete()	在特定的URL上对资源执行HTTP DELETE操作
exchange()	在URL上执行特定的HTTP方法，返回包含对象的ResponseEntity，这个对象是从响应体中映射得到的
execute()	在URL上执行特定的HTTP方法，返回一个从响应体映射得到的对象
getForEntity()	发送一个HTTP GET请求，返回的ResponseEntity包含了响应体所映射成的对象

getForObject()	发送一个HTTP GET请求，返回的请求体将映射为一个对象
headForHeaders()	发送HTTP HEAD请求，返回包含特定资源URL的HTTP头
optionsForAllow()	发送HTTP OPTIONS请求，返回对特定URL的Allow头信息
postForEntity()	POST数据到一个URL，返回包含一个对象的ResponseEntity，这个对象是从响应体中映射得到的
postForLocation()	POST数据到一个URL，返回新创建资源的URL
postForObject()	POST数据到一个URL，返回根据响应体匹配形成的对象
put()	PUT资源到特定的URL

16.4.2 GET资源

你可能意识到在表16.2中列出了两种执行GET请求的方法：getForObject()和getForEntity()。正如之前所描述的，每个方法又有三种形式的重载。三个getForObject()方法的签名如下：

```
<T> T getForObject(URI url, Class<T> responseType)
                         throws RestClientException;
<T> T getForObject(String url, Class<T> responseType,
                  Object... uriVariables)  throws RestClientException;
<T> T getForObject(String url, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

类似地，getForEntity()方法的签名如下：

```
<T> ResponseEntity<T> getForEntity(URI url, Class<T> responseType)
        throws RestClientException;
<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
        Object... uriVariables) throws RestClientException;
<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

除了返回类型，getForEntity()方法就是getForObject()方法的镜像。实际上，它们的工作方式大同小异。它们都执行根据URL检索资源的GET请求。它们都将资源根据responseType参数匹配为一

定的类型。唯一的区别在于getForObject()只返回所请求类型的对象，而getForEntity()方法会返回请求的对象以及响应相关的额外信息。

让我们首先看一下稍微简单的getForObject()方法。然后再看看如何使用getForEntity()方法来从GET响应中获取更多的信息。

16.4.3 检索资源

getForObject()方法是检索资源的合适选择。我们请求一个资源并按照所选择的Java类型接收该资源。作为getForObject()能够做什么的一个简单示例，让我们看一下fetchFacebookProfile()的另一个实现：

```
public Profile fetchFacebookProfile(String id) {
  RestTemplate rest = new RestTemplate();
  return rest.getForObject("http://graph.facebook.com/{spitter}",
      Profile.class, id);
}
```

在程序清单11.5中，fetchFacebookProfile()涉及十多行代码。通过使用RestTemplate，现在减少到了几行（如果我不是为了适应本书页面的边界，可能会更少）。

fetchFacebookProfile()首先构建了一个RestTemplate的实例（另一种可行的方式是注入实例）。接下来，它调用了getForObject()来得到Facebook个人信息。为了做到这一点，它要求结果是Profile对象。在接收到Profile对象后，该方法将其返回给调用者。

注意，在这个新版本的fetchFacebookProfile ()中，我们没有使用字符串连接来构建URL，而是利用了RestTemplate可以接受参数化URL这一功能。URL中的{id}占位符最终将会用方法的id参数来填充。getForObject()方法的最后一个参数是大小可变的参数列表，每个参数都会按出现顺序插入到指定URL的占位符中。

另外一种替代方案是将id参数放到Map中，并以id作为key，然后将这个Map作为最后一个参数传递给getForObject()：

```
public Spittle[] fetchFacebookProfile(String id) {
  Map<String, String> urlVariables = new HashMap<String, String();
  urlVariables.put("id", id);
  RestTemplate rest = new RestTemplate();
  return rest.getForObject("http://graph.facebook.com/{spitter}",
        Profile.class, urlVariables);
}
```

这里没有任何形式的JSON解析和对象映射。在表面之下，getForObject()为我们将响应体转换为对象。它实现这些需要依赖表16.1中所列的HTTP消息转换器，与带有@ResponseBody注解的Spring MVC处理方法所使用的一样。

这个方法也没有任何异常处理。这不是因为getForObject()不能抛出异常，而是因为它抛出的异常都是非检查型的。如果在getForObject()中有错误，将抛出非检查型RestClientException异常（或者它的一些子类）。如果愿意的话，你可以捕获它——但编译器不会强制你捕获它。

16.4.4 抽取响应的元数据

作为getForObject()的一个替代方案，RestTemplate还提供了getForEntity()。getForEntity()方法与getForObject()方法的工作很相似。getForObject()只返回资源（通过HTTP信息转换器将其转换为Java对象），getForEntity()会在ResponseEntity中返回相同的对象，而且ResponseEntity还带有关于响应的额外信息，如HTTP状态码和响应头。

我们可能想使用ResponseEntity所做的事就是获取响应头的一个值。例如，假设除了获取资源，还想要知道资源的最后修改时间。假设服务端在LastModified头部信息中提供了这个信息，我们可以这样像这样使用getHeaders()方法：

```
Date lastModified = new Date(response.getHeaders().getLastModified());
```

getHeaders()方法返回一个HttpHeaders对象，该对象提供了多个便利的方法来查询响应头，包括getLastModified()，它将返回从1970年1月1日开始的毫秒数。

除了getLastModified()，HttpHeaders还包含如下的方法来获取头信息：

```
public List<MediaType> getAccept() { ... }
public List<Charset> getAcceptCharset() { ... }
public Set<HttpMethod> getAllow() { ... }
public String getCacheControl() { ... }
public List<String> getConnection() { ... }
public long getContentLength() { ... }
public MediaType getContentType() { ... }
public long getDate() { ... }
public String getETag() { ... }
public long getExpires() { ... }
public long getIfNotModifiedSince() { ... }
public List<String> getIfNoneMatch() { ... }
public long getLastModified() { ... }
public URI getLocation() { ... }
public String getOrigin() { ... }
public String getPragma() { ... }
public String getUpgrade() { ... }
```

为了实现更通用的HTTP头信息访问，HttpHeaders提供了get()方法和getFirst()方法。两个方法都接受String参数来标识所需要的头信息。get()将会返回一个String值的列表，其中的每个值都是赋给该头部信息的，而getFirst()方法只会返回第一个头信息的值。

如果你对响应的HTTP状态码感兴趣，那么你可以调用getStatusCode()方法。例如，考虑下面这个获取Spittle对象的方法：

```
public Spittle fetchSpittle(long id) {
  RestTemplate rest = new RestTemplate();
  ResponseEntity<Spittle> response = rest.getForEntity(
      "http://localhost:8080/spittr-api/spittles/{id}",
      Spittle.class, id);
  if(response.getStatusCode() == HttpStatus.NOT_MODIFIED) {
    throw new NotModifiedException();
  }
  return response.getBody();
}
```

在这里，如果服务器响应304状态，这意味着服务器端的内容自从上一次请求之后再也没有修改。在这种情况下，将会抛出自定义的NotModifiedException异常来表明客户端应该检查它的缓存来获取Spittle。

16.4.5 PUT资源

为了对数据进行PUT操作，RestTemplate提供了三个简单的put()方法。就像其他的RestTemplate方法一样，put()方法有三种形式：

```
void put(URI url, Object request) throws RestClientException;
void put(String url, Object request, Object... uriVariables)
        throws RestClientException;
void put(String url, Object request, Map<String, ?> uriVariables)
        throws RestClientException;
```

按照它最简单的形式，put()接受一个java.net.URI，用来标识（及定位）要将资源发送到服务器上，另外还接受一个对象，这代表了资源的Java表述。

例如，以下展现了如何使用基于URI版本的put()方法来更新服务器上的Spittle资源：

```
public void updateSpittle(Spittle spittle) throws SpitterException {
  RestTemplate rest = new RestTemplate();
  String url = "http://localhost:8080/spittr-api/spittles/"
             + spittle.getId();
  rest.put(URI.create(url), spittle);
}
```

在这里，尽管方法签名很简单，但是使用java.net.URI作为参数的影响很明显。为了创建所更新Spittle对象的URL，我们要进行字符串拼接。

从getForObject()和getForEntity()方法中我们也看到了，使用基于String的其他put()方法能够为我们减少创建URI的不便。这些方法可以将URI指定为模板并对可变部分插入值。以下是使用基于String的put()方法重写的updateSpittle()：

```
public void updateSpittle(Spittle spittle) throws SpitterException {
  RestTemplate rest = new RestTemplate();
  rest.put("http://localhost:8080/spittr-api/spittles/{id}",
          spittle,  spittle.getId());
}
```

现在的URI使用简单的String模板来进行表示。当RestTemplate发送PUT请求时，URI模板将{id}部分用spittle.getId()方法的返回值来进行替换。就像getForObject()和getForEntity()一样，这个版本的put()方法最后一个参数是大小可变的参数列表，每一个值会出现按照顺序赋值给占位符变量。

你还可以将模板参数作为Map传递进来：

```
public void updateSpittle(Spittle spittle) throws SpitterException {
  RestTemplate rest = new RestTemplate();
  Map<String, String> params = new HashMap<String, String>();
  params.put("id", spittle.getId());
  rest.put("http://localhost:8080/spittr-api/spittles/{id}",
          spittle,  params);
}
```

当使用Map来传递模板参数时，Map条目的每个key值与URI模板中占位符变量的名字相同。

在所有版本的put()中，第二个参数都是表示资源的Java对象，它将按照指定的URI发送到服务器端。在本示例中，它是一个Spittle对象。RestTemplate将使用表16.1中的某个HTTP消息转换器将Spittle对象转换为一种表述形式，并在请求体中将其发送到服务器端。

对象将被转换成什么样的内容类型很大程度上取决于传递给put()方法的类型。如果给定一个String值，那么将会使用StringHttpMessageConverter：这个值直接被写到请求体中，内容类型设置为“text/plain”。如果给定一个MultiValueMap<String,String>，那么这个Map中的值将会被FormHttpMessageConverter以“application/x-www-form-urlencoded”的格式写到请求体中。

因为我们传递进来的是Spittle对象，所以需要一个能够处理任意对象的信息转换器。如果在类路径下包含Jackson 2库，那么MappingJacksonHttpMessageConverter将以application/json格式将Spittle写到请求中。

16.4.6 DELETE资源

当你不需要在服务端保留某个资源时，那么可能需要调用RestTemplate的delete()方法。就像PUT方法那样，delete()方法有三个版本，它们的签名如下：

```
void delete(String url, Object... uriVariables)
        throws RestClientException;
void delete(String url, Map<String, ?> uriVariables)
        throws RestClientException;
void delete(URI url) throws RestClientException;
```

很容易吧，delete()方法是所有RestTemplate方法中最简单的。你唯一要提供的就是要删除资源的URI。例如，为了删除指定ID的Spittle，你可以这样调用delete()：

```
public void deleteSpittle(long id) {
  RestTemplate rest = new RestTemplate();
  rest.delete(
      URI.create("http://localhost:8080/spittr-api/spittles/" + id));
}
```

这很简单，但在这里我们还是依赖字符串连接来创建URI对象。所以，我们再看一个更简单的delete()方法，它能够使得我们免于这些麻烦：

```
public void deleteSpittle(long id) {
  RestTemplate rest = new RestTemplate();
  rest.delete("http://localhost:8080/spittr-api/spittles/{id}", id));
}
```

你看，我感觉好多了。你呢？

现在我已经为你展现了最简单的RestTemplate方法，让我们看看RestTemplate最多样化的一组方法——它们能够支持HTTP POST请求。

16.4.7 POST资源数据

在16.2节的表中，你会看到RestTemplate有三个不同类型的方法来发送POST请求。当你再乘上每个方法的三个不同变种，那就是有九个方法来POST数据到服务器端。

这些方法中有两个的名字看起来比较类似。postForObject()和postForEntity()对POST请求的处理方式与发送GET请求的getForObject()和getForEntity()方法是类似的。另一个方法是getForLocation()，它是POST请求所特有的。

16.4.8 在POST请求中获取响应对象

假设你正在使用RestTemplate来POST一个新的Spitter对象到Spittr应用程序的REST API。因为这是一个全新的Spitter，服务端并（还）不知道它。因此，它还不是真正的REST资源，也没有URL。另外，在服务端创建之前，客户端并不知道Spitter的ID。

POST资源到服务端的一种方式是使用RestTemplate的postForObject()方法。postForObject()方法的三个变种签名如下：

```
<T> T postForObject(URI url, Object request, Class<T> responseType)
        throws RestClientException;
<T> T postForObject(String url, Object request, Class<T> responseType,
        Object... uriVariables) throws RestClientException;
<T> T postForObject(String url, Object request, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

在所有情况下，第一个参数都是资源要POST到的URL，第二个参数是要发送的对象，而第三个参数是预期返回的Java类型。在将URL作为String类型的两个版本中，第四个参数指定了URL变量（要么是可变参数列表，要么是一个Map）。

当POST新的Spitter资源到Spitter REST API时，它们应该发送到http://localhost:8080/spittr-api/spitters，这里会有一个应对POST请求的处理方法来保存对象。因为这个URL不需要URL参数，所以我们可以使用任何版本的postForObject()。但为了保持尽可能简单，我们可以这样调用：

```
public Spitter postSpitterForObject(Spitter spitter) {
  RestTemplate rest = new RestTemplate();
  return rest.postForObject("http://localhost:8080/spittr-api/spitters",
          spitter, Spitter.class);
}
```

postSpitterForObject()方法给定了一个新创建的Spitter对象，并使用postForObject()将其发送到服务器端。在响应中，它接收到一个Spitter对象并将其返回给调用者。

就像getForEntity()方法一样，你可能想得到请求带回来的一些元数据。在这种情况下，postForEntity()是更合适的方法。postForEntity()方法有着与postForObject()几乎相同的一组签名：

```
<T> ResponseEntity<T> postForEntity(URI url, Object request,
        Class<T> responseType) throws RestClientException;
<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Object... uriVariables)
        throws RestClientException;
<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Map<String, ?> uriVariables)
        throws RestClientException;
```

假设除了要获取返回的Spitter资源，还要查看响应中Location头信息的值。在这种情况下，你可以这样调用postForEntity()：

```
RestTemplate rest = new RestTemplate();
ResponseEntity<Spitter> response = rest.postForEntity(
    "http://localhost:8080/spittr-api/spitters",
    spitter, Spitter.class);
Spitter spitter = response.getBody();
URI url = response.getHeaders().getLocation();
```

与getForEntity()方法一样，postForEntity()返回一个ResponseEntity<T>对象。你可以调用这个对象的getBody()方法以获取资源对象（在本示例中是Spitter）。getHeaders()会给你一个HttpHeaders，通过它可以访问响应中返回的各种HTTP

头信息。这里，我们调用getLocation()来得到java.net.URI形式的Location头信息。

16.4.9 在POST请求后获取资源位置

如果要同时接收所发送的资源和响应头，postForEntity()方法是很便利的。但通常你并不需要将资源发送回来（毕竟，将其发送到服务器端是第一位的）。如果你真正需要的是Location头信息的值，那么使用RestTemplate的postForLocation()方法会更简单。

类似于其他的POST方法，postForLocation()会在POST请求的请求体中发送一个资源到服务器端。但是，响应不再是相同的资源对象，postForLocation()的响应是新创建资源的位置。它有如下三个方法签名：

```
URI postForLocation(String url, Object request, Object... uriVariables)
        throws RestClientException;
URI postForLocation(
        String url, Object request, Map<String, ?> uriVariables)
        throws RestClientException;
URI postForLocation(URI url, Object request) throws RestClientException;
```

为了展示postForLocation()，让我们再次POST一个Spitter。这次，我们希望在返回中包含资源的URL：

```
public String postSpitter(Spitter spitter) {
  RestTemplate rest = new RestTemplate();
  return rest.postForLocation(
          "http://localhost:8080/spittr-api/spitters",
          spitter).toString();
}
```

在这里，我们以String的形式将目标URL传递进来，还有要POST的Spitter对象（在本示例中没有URL参数）。在创建资源后，如果服务端在响应的Location头信息中返回新资源的URL，接下来postForLocation()会以String的格式返回该URL。

16.4.10 交换资源

到目前为止，我们已经看到RestTemplate的各种方法来GRT、PUT、DELETE以及POST资源。在它们之中，我们看到两个特殊的方法：getForEntity()和postForEntity()，这两个方法将结果资源包含在一个ResponseEntity对象中，通过这个对象我们可以得到响应头和状态码。

能够从响应中读取头信息是很有用的。但是如果你想在发送给服务端

的请求中设置头信息的话，怎么办呢？这就是RestTemplate的exchange()的用武之地。

像RestTemplate的其他方法一样，exchange()也重载为三个签名格式。一个使用java.net.URI来标识目标URL，而其他两个以String的形式传入URL并带有URL变量。如下所示：

```
<T> ResponseEntity<T> exchange(URI url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType)
        throws RestClientException;
<T> ResponseEntity<T> exchange(String url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType,
        Object... uriVariables) throws RestClientException;
<T> ResponseEntity<T> exchange(String url, HttpMethod method,
           HttpEntity<?> requestEntity, Class<T> responseType,
           Map<String, ?> uriVariables) throws RestClientException;
```

exchange()方法使用HttpMethod参数来表明要使用的HTTP动作。根据这个参数的值，exchange()能够执行与其他RestTemplate方法一样的工作。

例如，从服务器端获取Spitter资源的一种方式是使用RestTemplate的getForEntity()方法，如下所示：

```
ResponseEntity<Spitter> response = rest.getForEntity(
          "http://localhost:8080/spittr-api/spitters/{spitter}",
          Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

在下面的代码片段中，可以看到exchange()也可以完成这项任务：

```
ResponseEntity<Spitter> response = rest.exchange(
          "http://localhost:8080/spittr-api/spitters/{spitter}",
          HttpMethod.GET, null, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

通过传入HttpMethod.GET作为HTTP动作，我们会要求exchange()发送一个GET请求。第三个参数是用于在请求中发送资源的，但因为这是一个GET请求，它可以是null。下一个参数表明我们希望将响应转换为Spitter对象。最后一个参数用于替换URL模板中{spitter}占位符的值。

按照这种方式，exchange()与之前使用的getForEntity()是几乎相同的，但是，不同于getForEntity()——或getForObject()——exchange()方法允许在请求中设置头信息。接下来，我们不再给exchange()传递null，而是传入带有请求头信息的HttpEntity。

如果不指明头信息，exchange()对Spitter的GET请求会带有如下的头信息：

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/xml, text/xml, application/*+xml, application/json
Content-Length: 0
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

让我们看一下Accept头信息。Accept头信息表明它能够接受多种不同的XML内容类型以及application/json。这样服务器端在决定采用哪种格式返回资源时，就有很大的可选空间。假设我们希望服务端以JSON格式发送资源。在这种情况下，我们需要将“application/json”设置为Accept头信息的唯一值。

设置请求头信息是很简单的，只需构造发送给exchange()方法的HttpEntity对象即可，HttpEntity中包含承载头信息的MultiValueMap：

```
MultiValueMap<String, String> headers =
    new LinkedMultiValueMap<String, String>();
headers.add("Accept", "application/json");
HttpEntity<Object> requestEntity = new HttpEntity<Object>(headers);
```

在这里，我们创建了一个LinkedMultiValueMap并添加值为“application/json”的Accept头信息。接下来，我们构建了一个HttpEntity（使用Object泛型类型），将MultiValueMap作为构造参数传入。如果这是一个PUT或POST请求，我们需要为HttpEntity设置在请求体中发送的对象——对于GET请求来说，这是没有必要的。

现在，我们可以传入HttpEntity来调用exchange()：

```
ResponseEntity<Spitter> response = rest.exchange(
          "http://localhost:8080/spittr-api/spitters/{spitter}",
          HttpMethod.GET, requestEntity, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

表面上看，结果是一样的。我们得到了请求的Spitter对象。但在表面之下，请求将会带有如下的头信息发送：

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/json
Content-Length: 0
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

假设服务器端能够将Spitter序列化为JSON，响应体将会以JSON格式来进行表述。

16.5 小结

RESTful架构使用Web标准来集成应用程序，使得交互变得简单自然。系统中的资源采用URL进行标识，使用HTTP方法进行管理并且会以一种或多种适合客户端的方式来进行表述。

在本章中，我们看到了如何编写响应RESTful资源管理请求的Spring MVC控制器。借助参数化的URL模式并将控制器处理方法与特定的HTTP方法关联，控制器能够响应对资源的GET、POST、PUT以及DELETE请求。

为了响应这些请求，Spring能够将资源背后的数据以最适合客户端的形式展现。对于基于视图的响应，ContentNegotiatingViewResolver能够在多个视图解析器产生的视图中选择出最适合客户端期望内容类型的那一个。或者，控制器的处理方法可以借助@ResponseBody注解完全绕过视图解析，并使用信息转换器将返回值转换为客户端的响应。

REST API为客户端暴露了应用的功能，它们暴露功能的方式恐怕最原始的API设计者做梦都想不到。REST API的客户端通常是移动应用或运行在Web浏览器中的JavaScript。但是，Spring应用也可以借助RestTemplate来使用这些API。

REST只是应用间通信的方法之一，在下一章中，我们将会学习如何在Spring应用中借助消息实现异步通信。

第17章　Spring消息

本章内容：

- 异步消息简介
- 基于JMS的消息功能
- 使用Spring和AMQP发送消息
- 消息驱动的POJO

在星期五下午4点55分，再有几分钟你就可以开始休假了。现在，你的时间只够开车到机场赶上航班了。但是在你打包离开之前，你需要确定老板和同事了解你目前的工作进展，这样他们就可以在星期一继续完成你留下的工作。不过，你的一些同事已经提前离开过周末去了，而你的老板正在忙于开会。你该怎么办呢？

你可以给老板打电话，但是这样做就会因为一个不重要的状态报告而造成不必要的会议中断。或许你可以再坚持一会，等到会议结束。但是令人郁闷的是，你根本不知道会议还要持续多长时间，而你又要赶飞机。或者，你可以在他的显示器上留一个便条，不过要和其他的100个便条贴在一起。

要想既传达到你的工作状态又能赶上飞机，最有效的方式就是发送一封电子邮件给你的老板和同事，详述工作进展并且承诺给他们寄张明信片。你不知道他们在哪里，也不知道他们什么时候才能真正读到你的邮件。但是你知道，他们终究会回到他们的办公桌旁，阅读你的邮件。而此时，你正在赶往机场的路上。

有些时候，需要直接和某些人交谈。如果你受伤了，需要救护车，你可能会拿起电话——而不会给医院发电子邮件。不过，在通常情况下，发送消息就可以满足要求，并且跟直接通信相比更具有一些优势，例如可以让你继续你的假期。

在前面的一些章中，你看到了如何使用RMI、Hessian、Burlap、HTTP invoker和Web服务在应用程序之间进行通信。所有这些通信机制都是同步的，客户端应用程序直接与远程服务相交互，并且一直等到远程过程完成后才继续执行。

同步通信有它自己的适用场景。不过，对于开发者而言，这种通信方

式并不是应用程序之间进行交互的唯一方式。异步消息是一个应用程序向另一个应用程序间接发送消息的一种方式，这种方式无需等待对方的响应。相对于同步消息，异步消息具有多个优势，关于这一点你很快就会看到。

借助Spring，我们有多个实现异步消息的可选方案。在本章中，我们将会看到如何在Spring中使用Java消息服务（Java Message Service，JMS）和高级消息队列协议（Advanced Message Queuing Protocol，AMQP）发送和接收消息。除了基本的消息发送和接收之外，我们还会看到Spring对消息驱动POJO的支持，它是一种与EJB的消息驱动Bean（message-driven bean，MDB）类似的消息接收方式。

17.1　异步消息简介

与前面几章中介绍的远程调用机制以及REST接口类似，异步消息也是用于应用程序之间通信的。但是，在系统之间传递信息的方式上，它与其他机制有所不同。

像RMI和Hessian/Burlap这样的远程调用机制是同步的。如图17.1所示，当客户端调用远程方法时，客户端必须等到远程方法完成后，才能继续执行。即使远程方法不向客户端返回任何信息，客户端也要被阻塞直到服务完成。

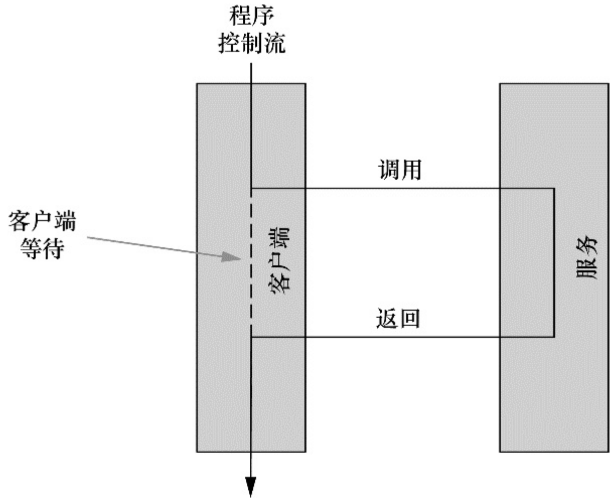

图17.1　如果通信是同步的，客户端必须等待服务完成

消息则是异步发送的，如图17.2所示，客户端不需要等待服务处理消息，甚至不需要等待消息投递完成。客户端发送消息，然后继续执行，这是因为客户端假定服务最终可以收到并处理这条消息。

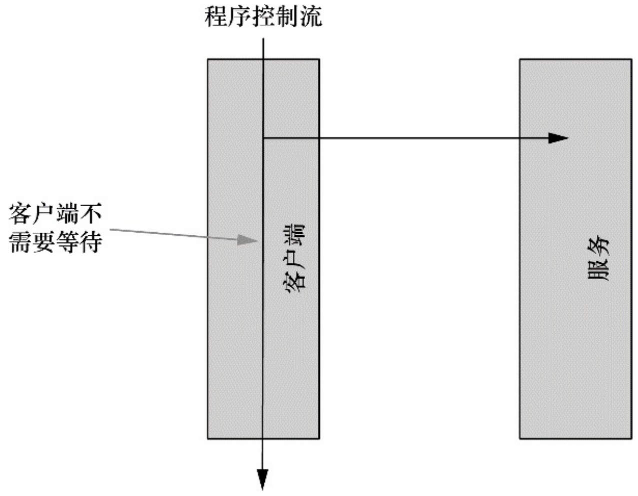

图17.2　异步通信是一种不需要等待的通信形式

相对于同步通信，异步通信具有多项优势，我们很快就会看到这些优点。但是首先，让我们看看如何异步发送消息。

17.1.1　发送消息

大多数人都使用过邮政服务。每天会有数百万信件、明信片和包裹交到邮递员手上，我们相信自己邮寄的东西会被送到目的地。世界实在是太大了，我们无法自己去运送这些东西，因此我们依赖邮政系统为我们运送。我们在信封上写明地址，贴张邮票，接着把它们投到信箱里，而不需要考虑信件如何到达目的地。

邮政服务的关键在于间接性。当奶奶的生日到来时，如果我们直接送给她一张贺卡，这非常不方便。我们必须留出几小时甚至是几天的时间去为她送生日贺卡，这取决于她住哪里。幸运的是，邮局可以将贺卡送到奶奶那里，而我们可以继续自己的生活。

与此类似，间接性也是异步消息的关键所在。当一个应用向另一个应用发送消息时，两个应用之间没有直接的联系。相反的是，发送方的应用程序会将消息交给一个服务，由服务确保将消息投递给接收方应用程序。

在异步消息中有两个主要的概念：消息代理（message broker）和目的地（destination）。当一个应用发送消息时，会将消息交给一个消息代理。消息代理实际上类似于邮局。消息代理可以确保消息被投递到指定的目的地，同时解放发送者，使其能够继续进行其他的业务。

当我们通过邮局邮递信件时，最重要的是要写上地址，这样邮局就可

以知道这封信应该被投递到哪里。与此类似，每条异步消息都带有一个目的地，目的地就好像一个邮箱，可以将消息放入这个邮箱，直到有人将它们取走。

但是，并不像信件地址那样必须标识特定的收件人或街道地址，消息中的目的地相对来说并不那么具体。目的地只关注消息应该从哪里获得——而不关心是由谁取走消息的。这种情况下，目的地就如同信件的地址为“本地居民”。

尽管不同的消息系统会提供不同的消息路由模式，但是有两种通用的目的地：队列（queue）和主题（topic）。每种类型都与特定的消息模型相关联，分别是点对点模型（队列）和发布/订阅模型（主题）。

点对点消息模型

在点对点模型中，每一条消息都有一个发送者和一个接收者，如图17.3所示。当消息代理得到消息时，它将消息放入一个队列中。当接收者请求队列中的下一条消息时，消息会从队列中取出，并投递给接收者。因为消息投递后会从队列中删除，这样就可以保证消息只能投递给一个接收者。

图17.3　消息队列对消息发送者和消息接收者进行了解耦。虽然队列可以有多个接收者，但是每一条消息只能被一个接收者取走

尽管消息队列中的每一条消息只被投递给一个接收者，但是并不意味着只能使用一个接收者从队列中获取消息。事实上，通常可以使用几个接收者来处理队列中的消息。不过，每个接收者都会处理自己所接收到的消息。

这与在银行排队等候类似。在等待时，我们可能注意到很多银行柜员都可以帮助我们处理金融业务。在柜员帮助客户完成业务后，她就空闲了，此时，她会要求排队等候的下一个人前来办理业务。如果我们排在队伍的最前边时，我们就会被叫到，然后由其中的一个空闲柜员来帮助我们处理业务，而其他的柜员则会帮助其他的银行客户。

从另一个角度看，我们在银行排队时，并不知道哪一个柜员会帮助我们办理业务。我们可以计算队伍中有多少人，与柜员的数目进行比

较，注意哪一个柜员业务办理速度最快，然后猜测会由哪一个柜员办理我们的业务。但是，一般情况下我们都会猜错，最终会由另一个柜员来办理。

同样，在点对点的消息中，如果有多个接收者监听队列，我们也无法知道某条特定的消息会由哪一个接收者处理。这种不确定性实际上有很多好处，因为我们只需要简单地为队列添加新的监听器就能提高应用的消息处理能力。

发布—订阅消息模型

在发布—订阅消息模型中，消息会发送给一个主题。与队列类似，多个接收者都可以监听一个主题。但是，与队列不同的是，消息不再是只投递给一个接收者，而是主题的所有订阅者都会接收到此消息的副本，如图17.4所示。

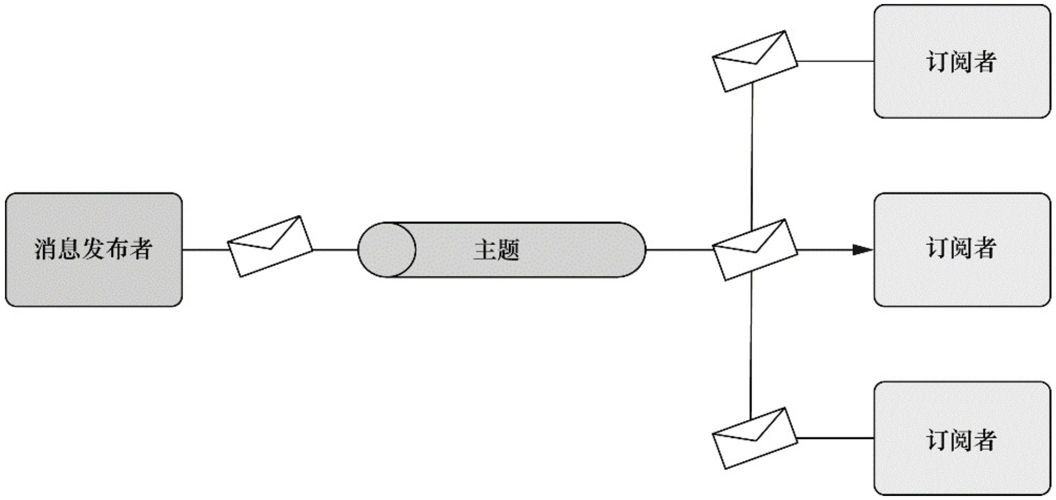

图17.4　与队列类似，主题可以将消息发送者与消息接收者进行解耦。与队列不同的是，主题消息可以发送给多个主题订阅者

正如它的名字所暗示的，发布—订阅消息模型与杂志发行商和杂志订阅者很相似。杂志（消息）出版后，发送给邮局，然后所有的订阅者都会收到杂志的副本。

杂志的类比就到此为至，因为对于异步消息来讲，发布者并不知道谁订阅了它的消息。发布者只知道它的消息要发送到一个特定的主题——而不知道有谁在监听这个主题。也就是说，发布者并不知道消息是如何被处理的。

现在，我们已经介绍了异步消息的基本概念，下面让我们看看它与同步RPC的对比。

17.1.2 评估异步消息的优点

虽然同步通信比较容易理解，建立起来也很简单，但是采用同步通信机制访问远程服务的客户端存在几个限制，最主要的是：

- 同步通信意味着等待。当客户端调用远程服务的方法时，它必须等待远程方法结束后才能继续执行。如果客户端与远程服务频繁通信，或者远程服务响应很慢，就会对客户端应用的性能带来负面影响。
- 客户端通过服务接口与远程服务相耦合。如果服务的接口发生变化，此服务的所有客户端都需要做相应的改变。
- 客户端与远程服务的位置耦合。客户端必须配置服务的网络位置，这样它才知道如何与远程服务进行交互。如果网络拓扑进行调整，客户端也需要重新配置新的网络位置。
- 客户端与服务的可用性相耦合。如果远程服务不可用，客户端实际上也无法正常运行。

虽然同步通信仍然有它的适用场景，但是在决定应用程序更适合哪种通信机制时，我们必须考量以上的这些缺点。如果这些限制正是你所担心的，那你可能很想知道异步通信是如何解决这些问题的。

无需等待

当使用JMS发送消息时，客户端不必等待消息被处理，甚至是被投递。客户端只需要将消息发送给消息代理，就可以确信消息会被投递给相应的目的地。

因为不需要等待，所以客户端可以继续执行其他任务。这种方式可以有效地节省时间，所以客户端的性能能够极大的提高。

面向消息和解耦

与面向方法调用的RPC通信不同，发送异步消息是以数据为中心的。这意味着客户端并没有与特定的方法签名绑定。任何可以处理数据的队列或主题订阅者都可以处理由客户端发送的消息，而客户端不必了解远程服务的任何规范。

位置独立

同步RPC服务通常需要网络地址来定位。这意味着客户端无法灵活地适应网络拓扑的改变。如果服务的IP地址改变了，或者服务被配置为

监听其他端口，客户端必须进行相应的调整，否则无法访问服务。

与之相反，消息客户端不必知道谁会处理它们的消息，或者服务的位置在哪里。客户端只需要了解需要通过哪个队列或主题来发送消息。因此，只要服务能够从队列或主题中获取消息即可，消息客户端根本不需要关注服务来自哪里。

在点对点模型中，可以利用这种位置的独立性来创建服务的集群。如果客户端不知道服务的位置，并且服务的唯一要求就是可以访问消息代理，那么我们就可以配置多个服务从同一个队列中接收消息。如果服务过载，处理能力不足，我们只需要添加一些新的服务实例来监听相同的队列就可以了。

在发布-订阅模型中，位置独立性会产生另一种有趣的效应。多个服务可以订阅同一个主题，接收相同消息的副本。但是每一个服务对消息的处理逻辑却可能有所不同。例如，假设我们有一组服务可以共同处理描述新员工信息的消息。一个服务可能会在工资系统中增加该员工，另一个服务则会将新员工增加到HR门户中，同时还有一个服务为新员工分配可访问系统的权限。每一个服务都基于相同的数据（都是从同一个主题接收的），但各自进行独立的处理。

确保投递

为了使客户端可以与同步服务通信，服务必须监听指定的IP地址和端口。如果服务崩溃了，或者由于某种原因无法使用了，客户端将不能继续处理。

但是，当发送异步消息时，客户端完全可以相信消息会被投递。即使在消息发送时，服务无法使用，消息也会被存储起来，直到服务重新可以使用为止。

现在，我们已经对异步消息的基础知识有所了解，接下来看一下如何将其付诸实施。首先，我们会使用JMS来发送和接收消息。

17.2 使用JMS发送消息

Java消息服务（Java Message Service，JMS）是一个Java标准，定义了使用消息代理的通用API。在JMS出现之前，每个消息代理都有私有的API，这就使得不同代理之间的消息代码很难通用。但是借助JMS，所有遵从规范的实现都使用通用的接口，这就类似于JDBC为

数据库操作提供了通用的接口一样。

Spring通过基于模板的抽象为JMS功能提供了支持，这个模板也就是JmsTemplate。使用JmsTemplate，能够非常容易地在消息生产方发送队列和主题消息，在消费消息的那一方，也能够非常容易地接收这些消息。Spring还提供了消息驱动POJO的理念：这是一个简单的Java对象，它能够以异步的方式响应队列或主题上到达的消息。

我们将会讨论Spring对JMS的支持，包括JmsTemplate和消息驱动POJO。但是在发送和接收消息之前，我们首先需要一个消息代理，它能够在消息的生产者和消费者之间传递消息。对Spring JMS的探索就从在Spring中搭建消息代理开始吧。

17.2.1 在Spring中搭建消息代理

ActiveMQ是一个伟大的开源消息代理产品，也是使用JMS进行异步消息传递的最佳选择。在我编写本书的时候，ActiveMQ的最新版本为5.9.1。在开始使用ActiveMQ之前，我们需要从http://activemq.apache.org下载二进制发行包。下载完ActiveMQ后，我们将其解压缩到本地硬盘中。在解压目录中，我们会找到文件activemq-core-5.9.1.jar。为了能够使用ActiveMQ的API，我们需要将此JAR文件添加到应用程序的类路径中。

在bin目录下，我们可以看到为各种操作系统所创建的对应子目录。在这些子目录下，我们可以找到用于启动ActiveMQ的脚本。例如，要在OS X下启动ActiveMQ，我们只需要在“bin/macosx”目录下运行activemq start。运行脚本后，ActiveMQ就准备好了，这时可以使用它作为消息代理。

创建连接工厂

在本章中，我们将了解如何采用不同的方式在Spring中使用JMS发送和接收消息。在所有的示例中，我们都需要借助JMS连接工厂通过消息代理发送消息。因为选择了ActiveMQ作为我们的消息代理，所以我们必须配置JMS连接工厂，让它知道如何连接到ActiveMQ。ActiveMQConnectionFactory是ActiveMQ自带的连接工厂，在Spring中可以使用如下方式进行配置：

```
<bean id="connectionFactory"
      class="org.apache.activemq.spring.ActiveMQConnectionFactory" />
```

默认情况下，ActiveMQConnectionFactory会假设ActiveMQ代理监听localhost的61616端口。对于开发环境来说，这没有什么问题，但是在生产环境下，ActiveMQ可能会在不同的主机和/端口上。如果是这样的话，我们可以使用brokerURL属性来指定代理的URL：

```
<bean id="connectionFactory"
      class="org.apache.activemq.spring.ActiveMQConnectionFactory"
      p:brokerURL="tcp://localhost:61616"/>
```

配置连接工厂还有另外一种方式，既然我们知道正在与ActiveMQ打交道，那我们就可以使用ActiveMQ自己的Spring配置命名空间来声明连接工厂（适用于ActiveMQ 4.1之后的所有版本）。首先，我们必须确保在Spring的配置文件中声明了amq命名空间：

```
<?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jms="http://www.springframework.org/schema/jms"
   xmlns:amq="http://activemq.apache.org/schema/core"
   xsi:schemaLocation="http://activemq.apache.org/schema/core
     http://activemq.apache.org/schema/core/activemq-core.xsd
     http://www.springframework.org/schema/jms
     http://www.springframework.org/schema/jms/spring-jms.xsd
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd">
   ...
  </beans>
```

现在我们就可以使用<amq:connectionFactory>元素声明连接工厂：

```
<amq:connectionFactory id="connectionFactory"
                       brokerURL="tcp://localhost:61616"/>
```

注意，<amq:connectionFactory>元素很明显是为ActiveMQ所准备的。如果我们使用不同的消息代理实现，它们不一定会提供Spring配置命名空间。如果没有提供的话，那我们就需要使用<bean>来装配连接工厂。

在本章的后续内容中，我们会多次使用connectionFactorybean，但是现在，我们只需要通过配置brokerURL属性来告知连接工厂消息代理的位置就足够了。在本例中，brokerURL属性中的URL指定连接工厂要连接到本地机器的61616端口（这个端口是ActiveMQ监听的默认端口）上的ActiveMQ。

声明**ActiveMQ**消息目的地

除了连接工厂外，我们还需要消息传递的目的地。目的地可以是一个

队列，也可以是一个主题，这取决于应用的需求。

不论使用的是队列还是主题，我们都必须使用特定的消息代理实现类在Spring中配置目的地bean。例如，下面的<bean>声明定义了一个ActiveMQ队列：

```
<bean id="queue"
      class="org.apache.activemq.command.ActiveMQQueue"
      c:_="spitter.queue" />
```

同样，下面的<bean>声明定义了一个ActiveMQ主题：

```
<bean id="topic"
      class="org.apache.activemq.command.ActiveMQTopic"
      c:_="spitter.queue" />
```

在第一个示例中，构造器指定了队列的名称，这样消息代理就能获知该信息，而在接下来示例中，名称则为spitter.topic。

与连接工厂相似的是，ActiveMQ命名空间提供了另一种方式来声明队列和主题。对于队列，我们可以使用<amq:quence>元素来声明：

```
<amq:queue id="spittleQueue" physicalName="spittle.alert.queue" />
```

如果是JMS主题，我们可以使用<amq:topic>元素来声明：

```
<amq:topic id="spittleTopic" physicalName="spittle.alert.topic" />
```

不管是哪种类型，都是借助physicalName属性指定消息通道的名称。

到此为止，我们已经看到了如何声明使用JMS所需的组件。现在我们已经准备好发送和接收消息了。为此，我们将使用Spring的JmsTemplate——Spring 对JMS支持的核心部分。但是首先，让我们先看看如果没有JmsTemplate，JMS是怎样使用的，以此了解JmsTemplate到底提供了些什么。

17.2.2 使用Spring的JMS模板

正如我们所看到的，JMS为Java开发者提供了与消息代理进行交互来发送和接收消息的标准API，而且几乎每个消息代理实现都支持JMS，因此我们不必因为使用不同的消息代理而学习私有的消息API。

虽然JMS为所有的消息代理提供了统一的接口，但是这种接口用起来并不是很方便。使用JMS发送和接收消息并不像拿一张邮票并贴在信封上那么简单。正如我们将要看到的，JMS还要求我们为邮递车加油（只是比喻的说法）。

处理失控的**JMS**代码

在10.3.1小节中，我向你展示了传统的JDBC代码在处理连接、语句、结果集和异常时是多么冗长和繁杂。遗憾的是，传统的JMS使用了类似的编程模型，如下面的程序清单所示。

程序清单**17.1**　使用传统的**JMS**（不使用**Spring**）发送消息

```
ConnectionFactory cf =
      new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
  conn = cf.createConnection();
  session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
  Destination destination = new ActiveMQQueue("spitter.queue");
  MessageProducer producer = session.createProducer(destination);
  TextMessage message = session.createTextMessage();

  message.setText("Hello world!");
  producer.send(message);                <— 发送消息
 } catch (JMSException e) {
  // handle exception?
} finally {
  try {
    if (session != null) {
      session.close();
    }
    if (conn != null) {
      conn.close();
    }
  } catch (JMSException ex) {
  }
}
```

再次声明这是一段失控的代码！就像JDBC示例一样，差不多使用了20行代码，只是为了发送一条“Hello world!”消息。实际上，其中只有几行代码是用来发送消息的，剩下的代码仅仅是为了发送消息而进行的设置。

接收端也没有好到哪里去，如下面的程序清单所示。

与程序清单17.1一样，程序清单17.2也是用一大段代码来实现如此简单的事情。如果我们逐行地比较，我们会发现它们几乎是完全一样的。如果查看上千个其他的JMS例子，我们会发现它们也是很相似的。只不过，其中一些会从JNDI中获取连接工厂，而另一些则是使用

主题代替队列。但是无论如何，它们都大致遵循相同的模式。

程序清单**17.2**　使用传统的**JMS**（不使用**Spring**）接收消息

```
ConnectionFactory cf =
      new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
  conn = cf.createConnection();
  conn.start();
  session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
  Destination destination =
      new ActiveMQQueue("spitter.queue");
   MessageConsumer consumer = session.createConsumer(destination);
   Message message = consumer.receive();
   TextMessage textMessage = (TextMessage) message;
   System.out.println("GOT A MESSAGE: " + textMessage.getText());
   conn.start();
 } catch (JMSException e) {
   // handle exception?
 } finally {
   try {
     if (session != null) {
       session.close();
     }
     if (conn != null) {
       conn.close();
     }
   } catch (JMSException ex) {
   }
 }
```

因为这些样板式代码，我们每次使用JMS时都要不断地做很多重复工作。更糟糕的是，你会发现我们在重复编写其他开发者的JMS代码。

我们已经在第10章看到了Spring的`JdbcTemplate`是如何处理失控的JDBC样板式代码的。现在，让我来介绍一下Spring的`JmsTemplate`如何对JMS的样板式代码实现相同的功能。

使用**JMS**模版

针对如何消除冗长和重复的JMS代码，Spring给出的解决方案是`JmsTemplate`。`JmsTemplate`可以创建连接、获得会话以及发送和接收消息。这使得我们可以专注于构建要发送的消息或者处理接收到的消息。

另外，`JmsTemplate`可以处理所有抛出的笨拙的`JMSException`异常。如果在使用`JmsTemplate`时抛出`JMSException`异常，`JmsTemplate`将捕获该异常，然后抛出一个非检查型异常，该异常是Spring自带的`JmsException`异常的子类。表17.1列出了标准的`JMSException`异常与Spring的非检查型异常之间的映射关系。

表17.1　Spring的JmsTemplate会捕获标准的JMSException异常，再以Spring的非检查型异常JmsException子类重新抛出

Spring（org.springframework.jms.*）	标准的JMS（javax.jms.*）
DestinationResolutionException	Spring特有的——当Spring无法解析目的地名称时抛出
IllegalStateException	IllegalStateException
InvalidClientIDException	InvalidClientIDException
InvalidDestinationException	InvalidSelectorException
InvalidSelectorException	InvalidSelectorException
JmsSecurityException	JmsSecurityException
ListenerExecutionFailedException	Spring特有的——当监听器方法执行失败时抛出
MessageConversionException	Spring特有的——当消息转换失败时抛出
MessageEOFException	MessageEOFException
MessageFormatException	MessageFormatException
MessageNotReadableException	MessageNotReadableException
MessageNotWriteableException	MessageNotWriteableException
ResourceAllocationException	ResourceAllocationException
SynchedLocalTransactionFailedException	Spring特有的——当同步的本地事务不

	能完成时抛出
TransactionInprogressException	TransactionInprogressException
TransactionRolledBackException	TransactionRolledBackException
UncategorizedJmsException	Spring特有的——当没有其他异常适用时抛出

对于JMS API来说，JMSException的确提供了丰富且具有描述性的子类集合，让我们更清楚地知道发生了什么错误。不过，所有的JMSException异常的子类都是检查型异常，因此必须要捕获。JmsTemplate为我们捕获这些异常，并重新抛出对应非检查型JMSException异常的子类。

为了使用JmsTemplate，我们需要在Spring的配置文件中将它声明为一个bean。如下的XML可以完成这项工作：

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate"
      c:_-ref="connectionFactory" />
```

因为JmsTemplate需要知道如何连接到消息代理，所以我们必须为connection-Factory属性设置实现了JMS的ConnectionFactory接口的bean引用。在这里，我们使用在12.2.1小节中所声明的connectionFactorybean引用来装配该属性。

这就是配置JmsTemplate所需要做的所有工作——现在JmsTemplate已经准备好了。让我们开始发送消息吧！

发送消息

在我们想建立的Spittr应用程序中，其中有一个特性就是当创建Spittle的时候提醒其他用户（或许是通过E-mail）。我们可以在增加Spittle的地方直接实现该特性。但是搞清楚发送提醒给谁以及实际发送这些提醒可能需要一段时间，这会影响到应用的性能。当增加一个新的Spittle时，我们希望应用是敏捷的，能够快速做出响应。

与其在增加Spittle时浪费时间发送这些信息，不如对该项工作进行排队，在响应返回给用户之后再处理它。与直接发送消息给其他用户所

花费的时间相比，发送消息给队列或主题所花费的时间是微不足道的。

为了在Spittle创建的时候异步发送spittle提醒，让我们为Spittr应用引入AlertService：

```
package com.habuma.spittr.alerts;
import com.habuma.spittr.domain.Spittle;

public interface AlertService {
  void sendSpittleAlert(Spittle spittle);
}
```

正如我们所看到的，AlertService是一个接口，只定义了一个操作—— sendSpittleAlert()。

如程序清单17.3所示，AlertServiceImpl实现了AlertService接口，它使用JmsOperation（JmsTemplate所实现的接口）将Spittle对象发送给消息队列，而队列会在稍后得到处理。

程序清单**17.3**　使用**JmsTemplate**发送一个**Spittle**

```
package com.habuma.spittr.alerts;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsOperations;
import org.springframework.jms.core.MessageCreator;
import com.habuma.spittr.domain.Spittle;

public class AlertServiceImpl implements AlertService {

  private JmsOperations jmsOperations;

  @Autowired
  public AlertServiceImpl(JmsOperations jmsOperatons) {  <—— 注入 JMS 模板
     this.jmsOperations = jmsOperations;
  }

  public void sendSpittleAlert(final Spittle spittle) {

    jmsOperations.send(                                    <—— 发送消息
       "spittle.alert.queue",                <—— 指定目的地
       new MessageCreator() {
        public Message createMessage(Session session)
                throws JMSException {
          return session.createObjectMessage(spittle);      <—— 创建消息
        }
       }
    );
  }
}
```

JmsOperations的send()方法的第一个参数是JMS目的地名称，标识消息将发送给谁。当调用send()方法时，JmsTemplate将负责获得JMS连接、会话并代表发送者发送消息（如图17.5所示）。

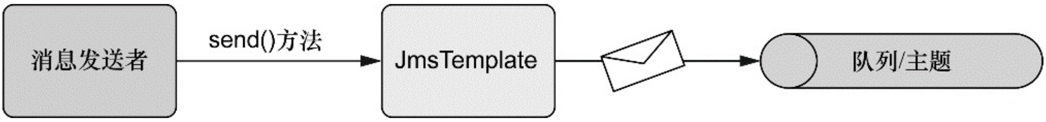

图17.5　JmsTemplate代表发送者来负责处理发送消息的复杂过程

我们使用MessageCreator（在这里的实现是作为一个匿名内部类）来构造消息。在MessageCreator的createMessage()方法中，我们通过session创建了一个对象消息：传入一个Spittle对象，返回一个对象消息。

就是这么简单！注意，sendSpittleAlert()方法专注于组装和发送消息。在这里没有连接或会话管理的代码，JmsTemplate帮我们处理了所有的相关事项，而且我们也不需要捕获JMSException异常。JmsTemplate将捕获抛出的所有JMSException异常，然后重新抛出表17.1所列的某一种非检查型异常。

设置默认目的地

在程序清单17.3中，我们明确指定了一个目的地，在send()方法中将Spittle消息发向此目的地。当我们希望通过程序选择一个目的地时，这种形式的send()方法很适用。但是在AlertServiceImpl案例中，我们总是将Spittle消息发给相同的目的地，所以这种形式的send()方法并不能带来明显的好处。

与其每次发送消息时都指定一个目的地，不如我们为JmsTemplate装配一个默认的目的地：

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate"
      c:_-ref="connectionFactory"
      p:defaultDestinationName="spittle.alert.queue" />
```

在这里，将目的地的名称设置为spittle.alert.queue，但它只是一个名称：它并没有说明你所处理的目的地是什么类型。如果已经存在该名称的队列或主题的话，就会使用已有的。如果尚未存在的话，将会创建一个新的目的地（通常会是队列）。但是，如果你想指定要创建的目的地类型的话，那么你可以将之前创建的队列或主题的目的地bean装配进来：

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate"
      c:_-ref="connectionFactory"
      p:defaultDestination-ref="spittleTopic" />
```

现在，调用JmsTemplate的send()方法时，我们可以去除第一个参数了：

```
jmsOperations.send(
  new MessageCreator() {
  ...
  }
);
```

这种形式的send()方法只需要传入一个MessageCreator。因为希望消息发送给默认目的地，所以我们没有必要再指定特定的目的地。

在调用send()方法时，我们不必再显式指定目的地能够让任务得以简化。但是如果我们使用消息转换器的话，发送消息会更加简单。

在发送时，对消息进行转换

除了send()方法，JmsTemplate还提供了convertAndSend()方法。与send()方法不同，convertAndSend()方法并不需要MessageCreator作为参数。这是因为convertAndSend()会使用内置的消息转换器（message converter）为我们创建消息。

当我们使用convertAndSend()时，sendSpittleAlert()可以减少到方法体中只包含一行代码：

```
public void sendSpittleAlert(Spittle spittle) {
  jmsOperations.convertAndSend(spittle);
}
```

就像变魔术一样，Spittle会在发送之前转换为Message。不过就像所有的魔术一样，JmsTemplate内部会进行一些处理。它使用一个MessageConverter的实现类将对象转换为Message。

MessageConverter是Spring定义的接口，只有两个需要实现的方法：

```
public interface MessageConverter {
  Message toMessage(Object object, Session session)
                  throws JMSException, MessageConversionException;
  Object fromMessage(Message message)
                  throws JMSException, MessageConversionException;
}
```

尽管这个接口实现起来很简单，但我们通常并没有必要创建自定义的实现。Spring已经提供了多个实现，如表17.2所示。

表17.2　Spring为通用的转换任务提供了多个消息转换器

（所有的消息转换器都位于**org.springframework.jms.support.converter**包中）

消息转换器	功　　能
MappingJacksonMessageConverter	使用Jackson JSON库实现消息与JSON格式之间的相互转换
MappingJackson2MessageConverter	使用Jackson 2 JSON库实现消息与JSON格式之间的相互转换
MarshallingMessageConverter	使用JAXB库实现消息与XML格式之间的相互转换
SimpleMessageConverter	实现String与TextMessage之间的相互转换，字节数组与BytesMessage之间的相互转换，Map与MapMessage之间的相互转换以及Serializable对象与ObjectMessage之间的相互转换

默认情况下，JmsTemplate在convertAndSend()方法中会使用SimpleMessage Converter。但是通过将消息转换器声明为bean并将其注入到JmsTemplate的messageConverter属性中，我们可以重写这种行为。例如，如果你想使用JSON消息的话，那么可以声明一个MappingJacksonMessageConverter bean：

```
<bean id="messageConverter"
      class="org.springframework.jms.support.converter.
                           ➥ MappingJacksonMessageConverter" />
```

然后，我们可以将其注入到JmsTemplate中，如下所示：

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate"
      c:_-ref="connectionFactory"
      p:defaultDestinationName="spittle.alert.queue"
      p:messageConverter-ref="messageConverter" />
```

各个消息转换器可能会有额外的配置，进而实现转换过程的细粒度控制。例如，MappingJacksonMessageConverter能够让我们配置转码以及自定义Jackson ObjectMapper。可以查阅每个消息转换器的JavaDoc以了解如何更加细粒度地配置它们。

接收消息

现在我们已经了解了如何使用JmsTemplate发送消息。但如果我们是接收端，那要怎么办呢？JmsTemplate是不是也可以接收消息呢？

没错，的确可以。事实上，使用JmsTemplate接收消息甚至更简单，我们只需要调用JmsTemplate的receive()方法即可，如程序清单12.4所示。

当调用JmsTemplate的receive()方法时，JmsTemplate会尝试从消息代理中获取一个消息。如果没有可用的消息，receive()方法会一直等待，直到获得消息为止。图17.6展示了这个交互过程。

程序清单**17.4**　使用**JmsTemplate**接收消息

```
public Spittle receiveSpittleAlert() {
  try {
    ObjectMessage receivedMessage =
        (ObjectMessage) jmsOperations.receive();          <-- 接收消息

    return (Spittle) receivedMessage.getObject();          <-- 获得对象
  } catch (JMSException jmsException) {
    throw JmsUtils.convertJmsAccessException(jmsException); <-- 抛出转换后的异常
  }
}
```

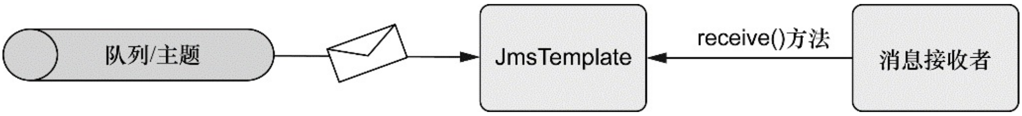

图17.6　使用JmsTemplate从主题或队列中接收消息的时候，只需要简单地调用receive()方法。JmsTemplate会处理其他的事情

因为我们知道Spittle消息是作为一个对象消息来发送的，所以它可以在到达后转型为ObjectMessage。然后，我们调用getObject()方法把ObjectMessage转换为Spittle对象并返回此对象。

但是这里存在一个问题，我们不得不对可能抛出的JMSException进行处理。正如我已经提到的，JmsTemplate可以很好地处理抛出的JmsException检查型异常，然后把异常转换为Spring非检查型异常JmsException并重新抛出。但是它只对调用JmsTemplate的方法时才适用。JmsTemplate无法处理调用ObjectMessage的getObject()方法时所抛出的JMSException异常。

因此，我们要么捕获JMSException异常，要么声明本方法抛出JMSException异常。为了遵循Spring规避检查型异常的设计理念，

我们不建议本方法抛出JMSException异常，所以我们选择捕获该异常。在catch代码块中，我们使用Spring中JmsUtils的convertJmsAccessException()方法把检查型异常JMSException转换为非检查型异常JmsException。这其实是在其他场景中由JmsTemplate为我们做的事情。

在receiveSpittleAlert()方法中，我们可以改善的一点就是使用消息转换器。在convertAndSend()中，我们已经看到了如何将对象转换为Message。不过，它们还可以用在接收端，也就是使用JmsTemplate的receiveAndConvert()：

```
public Spittle retrieveSpittleAlert() {
  return (Spittle) jmsOperations.receiveAndConvert();
}
```

现在，没有必要将Message转换为ObjectMessage，也没有必要通过调用getObject()来获取Spittle，更无需担心检查型的JMSException异常。这个新的retrieve SpittleAlert()简洁了许多。但是，依然还有一个很小且不容易察觉的问题。

使用JmsTemplate接收消息的最大缺点在于receive()和receiveAndConvert()方法都是同步的。这意味着接收者必须耐心等待消息的到来，因此这些方法会一直被阻塞，直到有可用消息（或者直到超时）。同步接收异步发送的消息，是不是感觉很怪异？

这就是消息驱动POJO的用武之处。让我们看看如何使用能够响应消息的组件异步接收消息，而不是一直等待消息的到来。

17.2.3 创建消息驱动的POJO

在学校时的一个暑假期间，我得到了在黄石国家公园工作的机会。这个工作并不是公园巡逻者或者开关老忠实泉（Old Faithful）这样的高级工作，而是在老忠实泉酒店进行更换床单、清理卫生间以及打扫地板等家务工作。虽然不是很吸引人，但至少我是在这个世界上最美丽的地方工作。

每天工作之后，我都到当地的邮局看看是否有我的邮件。我已经离家好几个星期了，所以能收到学校朋友的来信或者明信片是一件非常美好的事情。我没有自己的邮箱，所以必须走着去邮局，并询问坐在柜台后的工作人员是否有我的邮件。接着就是开始等待。

要知道，柜台后的那个人大约有195岁[1]了。像他这个岁数的人，走

动起来很费时间。他从椅子上站起来，慢慢走过地板，消失在隔墙后。过了一会儿，他出现了，慢慢回到柜台，坐到椅子上，然后看着我说："今天没有邮件"。

JmsTemplate的receive()方法与这个上了年纪的邮局雇员很像。当我们调用receive()方法时，JmsTemplate会查看队列或主题中是否有消息，直到收到消息或者等待超时才会返回。这期间，应用无法处理任何事情，只能等待是否有消息。如果应用能够继续进行其他业务处理，当消息到达时再去通知它，不是更好吗？

EJB2规范的一个重要内容是引入了消息驱动bean（message-driven bean，MDB）。MDB是可以异步处理消息的EJB。换句话说，MDB将JMS目的地中的消息作为事件，并对这些事件进行响应。而与之相反的是，同步消息接收者在消息可用前会一直处于阻塞状态。

MDB是EJB中的一个亮点。即使那些狂热的EJB反对者也认为MDB可以优雅地处理消息。EJB 2 MDB的唯一缺点是它们必须要实现java.ejb.MessageDriven-Bean。此外，它们还必须实现一些EJB生命周期的回调方法。简而言之，EJB 2 MDB 不是纯的POJO。

在EJB 3规范中，MDB进一步简化了，使其更像POJO。我们不再需要实现MessageDrivenBean接口，而是实现更通用的javax.jms.MessageListener接口，并使用@MessageDriven注解标注MDB。

Spring 2.0提供了它自己的消息驱动bean来满足异步接收消息的需求，这种形式与EJB 3的MDB很相似。在本节中，我们将学习到Spring是如何使用消息驱动POJO（我们将其简称为MDP）来支持异步接收消息的。

创建消息监听器

如果使用EJB的消息驱动模型来创建Spittle的提醒处理器，我们需要使用@MessageDriven注解进行标注。即使它不是严格要求的，但EJB规范还是建议MDB实现MessageListener接口。Spittle的提醒处理器最终可能是这样的：

```
@MessageDriven(mappedName="jms/spittle.alert.queue")
public class SpittleAlertHandler implements MessageListener {
  @Resource
  private MessageDrivenContext mdc;

  public void onMessage(Message message) {
     ...
  }
}
```

想象一下，如果消息驱动组件不需要实现MessageListener接口，世界将是多么的简单。在这里，天是蔚蓝的，鸟儿唱着我们喜欢的歌，我们不再需要实现onMessage()方法或者注入Messge DrivenContext。

好吧，可能EJB 3规范所要求的MDB也算不上太麻烦。但是事实上，SpittleAlertHandler的EJB 3实现太依赖于EJB的消息驱动API，并不是我们所希望的POJO。理想情况下，我们希望提醒处理器能够处理消息，但是不用编码，就好像它知道应该做什么。

Spring提供了以POJO的方式处理消息的能力，这些消息来自于JMS的队列或主题中。例如，基于POJO实现SpittleAlertHandler就足以做到这一点。

程序清单**17.5　Spring MDP**异步接收和处理消息

```
package com.habuma.spittr.alerts;
import com.habuma.spittr.domain.Spittle;

public class SpittleAlertHandler {

  public void handleSpittleAlert(Spittle spittle) {        <— 处理方法
     // ... implementation goes here...
  }
}
```

虽然改变天空的颜色和训练鸟儿歌唱超出了Spring的范围，但程序清单17.5所展示的现实与我描绘的理想世界非常接近。我们稍后会编写handleSpittleAlert()方法的具体内容。现在，程序清单 17.5所展示的SpittleAlertHandler没有任何JMS的痕迹。从任意一个角度观察，它都是一个纯粹的POJO。它仍然可以像EJB那样处理消息，只不过它还需要一些Spring的配置。

配置消息监听器

为POJO赋予消息接收能力的诀窍是在Spring中把它配置为消息监听器。Spring的jms命名空间为我们提供了所需要的一切。首先，让我们先把处理器声明为bean：

```
<bean id="spittleHandler"
      class="com.habuma.spittr.alerts.SpittleAlertHandler" />
```

然后，为了把SpittleAlertHandler转变为消息驱动的POJO，我们需要把这个bean声明为消息监听器：

```
<jms:listener-container connection-factory="connectionFactory">
  <jms:listener destination="spitter.alert.queue"
       ref="spittleHandler" method="handleSpittleAlert" />
</jms:listener-container>
```

在这里，我们在消息监听器容器中包含了一个消息监听器。消息监听器容器（message listener container）是一个特殊的bean，它可以监控JMS目的地并等待消息到达。一旦有消息到达，它取出消息，然后把消息传给任意一个对此消息感兴趣的消息监听器。如图17.7展示了这个交互过程。

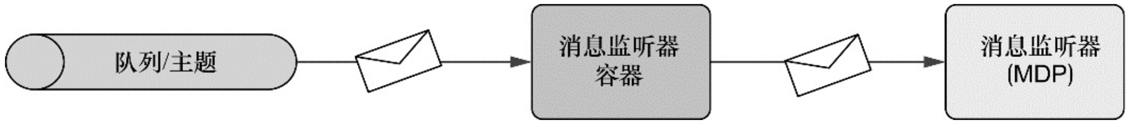

图17.7　消息监听器容器监听队列和主题。
当消息到达时，消息将转给消息监听器（例如消息驱动的POJO）

为了在Spring中配置消息监听器容器和消息监听器，我们使用了Spring jms命名空间中的两个元素。<jms:listener-container>中包含了<jms:listener>元素。这里的connection-factory属性配置了对connectionFactory的引用，容器中的每个<jms:listener>都使用这个连接工厂进行消息监听。在本示例中，connection-factory属性可以移除，因为该属性的默认值就是connectionFactory。

对于<jms:listener>元素，它用于标识一个bean和一个可以处理消息的方法。为了处理Spittle提醒消息，ref元素引用了spittleHandler bean。当消息到达spitter.alert.queue队列（通过destination属性配置）时，spittleHandlerbean的handleSpittleAlert()方法（通过method属性指定的）会被触发。

值得一提的是，如果ref属性所标示的bean实现了MessageListener，那就没有必要再指定method属性了，默认就会调用onMessage()方法。

17.2.4　使用基于消息的RPC

在第15章中，我们展示了Spring把bean的方法暴露为远程服务以及从客户端向这些服务发起调用的几种方式。在本章，我们学习了如何通过队列和主题在应用程序之间发送消息。现在我们将了解一下如何使用JMS作为传输通道来进行远程调用。

为了支持基于消息的RPC，Spring提供了`JmsInvokerServiceExporter`，它可以把bean导出为基于消息的服务；为客户端提供了`JmsInvokerProxyFactoryBean`来使用这些服务。

让我们回顾一下第15章，Spring提供了多种方式把bean导出为远程服务。我们使用`RmiServiceExporter`把bean导出为RMI服务，使用`HessianExporter`和`BurlapExporter`导出为基于HTTP的Hessian和Burlap服务，还使用`HttpInvoker Service Exporter`创建基于HTTP的HTTP invoker服务。但Spring还提供了一种在第15章中我们未探讨的服务导出器。

导出基于**JMS**的服务

`JmsInvokerServiceExporter`很类似于其他的服务导出器。事实上，`JmsInvoker-ServiceExporter`与`HttpInvokerServiceExporter`在名称上有某种对称型。如果`HttpInvokerServiceExporter`可以导出基于HTTP通信的服务，那么`JmsInvoker-ServiceExporter`就应该可以导出基于JMS的服务。

为了演示`JmsInvokerServiceExporter`是如何工作的，考虑如下的`AlertServiceImpl`。

程序清单**17.6　AlertServiceImpl**是一个处理**JMS**消息的**POJO**，但是不依赖于**JMS**

```java
package com.habuma.spittr.alerts;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.stereotype.Component;
import com.habuma.spittr.domain.Spittle;

@Component("alertService")
public class AlertServiceImpl implements AlertService {

   private JavaMailSender mailSender;
   private String alertEmailAddress;

  public AlertServiceImpl(JavaMailSender mailSender,
                          String alertEmailAddress) {
    this.mailSender = mailSender;
    this.alertEmailAddress = alertEmailAddress;
  }
  public void sendSpittleAlert(final Spittle spittle) {     <-- 发送 Spittle 提醒
     SimpleMailMessage message = new SimpleMailMessage();
    String spitterName = spittle.getSpitter().getFullName();
    message.setFrom("noreply@spitter.com");
    message.setTo(alertEmailAddress);
    message.setSubject("New spittle from " + spitterName);
    message.setText(spitterName + " says: " + spittle.getText());
    mailSender.send(message);
  }

}
```

我们现在不要过于关注sendSpittleAlert()方法的细节。在第19章，我们将会继续探讨如何使用Spring发送E-mail。现在，我们需要关注的重点在于AlertServiceImpl是一个简单的POJO，没有任何迹象标示它要用来处理JMS消息。它只是实现了简单的AlertService接口，该接口如下所示：

```java
package com.habuma.spittr.alerts;
import com.habuma.spittr.domain.Spittle;
public interface AlertService {
  void sendSpittleAlert(Spittle spittle);
}
```

正如我们所看到的，AlertServiceImpl使用了@Component注解来标注，所以它会被Spring自动发现并注册为Spring应用上下文中ID为alertService的bean。在配置JmsInvokerServiceExporter时，我们将引用这个bean：

```xml
<bean id="alertServiceExporter"
    class="org.springframework.jms.remoting.JmsInvokerServiceExporter"
    p:service-ref="alertService"
    p:serviceInterface="com.habuma.spittr.alerts.AlertService" />
```

这个bean的属性描述了导出的服务应该是什么样子的。service属性设置为alertServicebean的引用，它是远程服务的实现。同时，serviceInterface属性设置为远程服务对外提供接口的全限定类名。

导出器的属性并没有描述服务如何基于JMS通信的细节。但好消息是JmsInvokerServiceExporter可以充当JMS监听器。因此，我们使用<jms:listenercontainer>元素配置它：

```
<jms:listener-container connection-factory="connectionFactory">
  <jms:listener destination="spitter.alert.queue"
                ref="alertServiceExporter" />
</jms:listener-container>
```

我们为JMS监听器容器指定了连接工厂，所以它能够知道如何连接消息代理，而<jms:listener>声明指定了远程消息的目的地。

使用基于**JMS**的服务

这时候，基于JMS的提醒服务已经准备好了，等待队列中名字为spitter.alert.queue的RPC消息到达。在客户端，JmsInvokerProxyFactoryBean用来访问服务。

JmsInvokerProxyFactoryBean很类似于我们在第15章中所讨论的其他远程代理工厂bean。它隐藏了访问远程服务的细节，并提供一个易用的接口，通过该接口客户端与远程服务进行交互。与代理RMI服务或HTTP服务的最大区别在于，JmsInvokerProxy-FactoryBean代理了通过JmsInvokerServiceExporter所导出的JMS服务。

为了使用提醒服务，我们可以像下面那样配置JmsInvokerProxyFactoryBean：

```
<bean id="alertService"
  class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean"
  p:connectionFactory-ref="connectionFactory"
  p:queueName="spittle.alert.queue"
  propp:serviceInterface="com.habuma.spittr.alerts.AlertService" />
```

connectionFactory和queryName属性指定了RPC消息如何被投递——在这里，也就是在给定的连接工厂中，我们所配置的消息代理里面名为spitter.alert.queue的队列。对于serviceInterface，指定了代理应该通过AlertService接口暴露功能。

多年来，JMS一直是Java应用中主流的消息解决方案。但是对于Java和Spring开发者来说，JMS并不是唯一的消息可选方案。在过去的几年中，高级消息队列协议（Advanced Message Queuing Protocol，AMQP）得到了广泛的关注。因此，Spring也为通过AMQP发送消息提供了支持，这就是我们下面要讲解的内容。

17.3　使用AMQP实现消息功能

你可能会疑惑为什么还需要另外一个消息规范。难道JMS还不够好吗？AMQP提供了哪些JMS所不具备的特性呢？

实际上，AMQP具有多项JMS所不具备的优势。首先，AMQP为消息定义了线路层（wire-level protocol）的协议，而JMS所定义的是API规范。JMS的API协议能够确保所有的实现都能通过通用的API来使用，但是并不能保证某个JMS实现所发送的消息能够被另外不同的JMS实现所使用。而AMQP的线路层协议规范了消息的格式，消息在生产者和消费者间传送的时候会遵循这个格式。这样AMQP在互相协作方面就要优于JMS——它不仅能跨不同的AMQP实现，还能跨语言和平台。[2]

相比JMS，AMQP另外一个明显的优势在于它具有更加灵活和透明的消息模型。使用JMS的话，只有两种消息模型可供选择：点对点和发布-订阅。这两种模型在AMQP当然都是可以实现的，但AMQP还能够让我们以其他的多种方式来发送消息，这是通过将消息的生产者与存放消息的队列解耦实现的。

Spring AMQP是Spring框架的扩展，它能够让我们在Spring应用中使用AMQP风格的消息。稍后可以看到，Spring AMQP提供了一个API，借助这个API，我们能够以非常类似于Spring JMS抽象的形式来使用AMQP。这意味着，我们在本章之前所学习的JMS内容能够帮助你理解如何使用Spring AMQP来发送和接收消息。

我们稍后就会介绍如何使用Spring AMQP，但是在深入学习如何在Spring中发送和接收消息之前，首先看一下到底是什么让AMQP如此引人关注。

17.3.1　AMQP简介

简单回忆一下JMS的消息模型，可能会有助于理解AMQP的消息模型。在JMS中，有三个主要的参与者：消息的生产者、消息的消费者以及在生产者和消费者之间传递消息的通道（队列或主题）。JMS消息模型中的关键元素在图17.3和图17.4中进行了描述。

在JMS中，通道有助于解耦消息的生产者和消费者，但是这两者依然会与通道相耦合。生产者会将消息发布到一个特定的队列或主题上，消费者从特定的队列或主题上接收这些消息。通道具有双重责任，也

就是传递数据以及确定这些消息该发送到什么地方，队列的话会使用点对点算法发送，主题的话就使用发布-订阅的方式。

与之不同的是，AMQP的生产者并不会直接将消息发布到队列中。AMQP在消息的生产者以及传递信息的队列之间引入了一种间接的机制：Exchange。这种关系如图17.8所示。

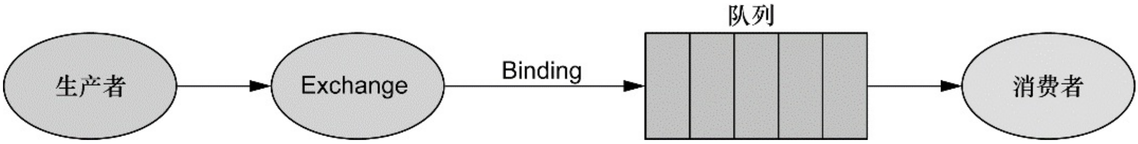

图17.8　在AMQP中，通过引入处理信息路由的Exchange，消息的生产者 与消息队列之间实现了解耦

可以看到，消息的生产者将信息发布到一个Exchange。Exchange会绑定到一个或多个队列上，它负责将信息路由到队列上。信息的消费者会从队列中提取数据并进行处理。

图17.8所没有展现出来的一点是Exchange不是简单地将消息传递到队列中，并不仅仅是一种穿透（pass-through）机制。AMQP定义了四种不同类型的Exchange，每一种都有不同的路由算法，这些算法决定了是否要将信息放到队列中。根据Exchange的算法不同，它可能会使用消息的*routing key*和/或参数，并将其与Exchange和队列之间binding的routing key和参数进行对比。（routing key可以大致理解为Email的收件人地址，指定了预期的接收者。）如果对比结果满足相应的算法，那么消息将会路由到队列上。否则的话，将不会路由到队列上。

四种标准的AMQP Exchange如下所示：

- Direct：如果消息的routing key与binding的routing key直接匹配的话，消息将会路由到该队列上；
- Topic：如果消息的routing key与binding的routing key符合通配符匹配的话，消息将会路由到该队列上；
- Headers：如果消息参数表中的头信息和值都与bingding参数表中相匹配，消息将会路由到该队列上；
- Fanout：不管消息的routing key和参数表的头信息/值是什么，消息将会路由到所有队列上。

借助这四种类型的Exchange，很容易就能想到我们可以定义任意数量的路由模式，而不再仅限于点对点和发布-订阅的方式。[3]好消息是，当发送和接收消息的时候，所涉及的路由算法对于如何编写消息

的生产者和消费者并没有什么影响。简单来讲，生产者将信息发送给Exchange并带有一个routing key，消费者从队列中获取消息。

我们已经快速了解了AMQP消息的基本知识——此时应该已经能够理解我们接下来所要介绍的如何使用Spring发送和接收消息。但是，我建议你更深入的学习一下AMQP，可以阅读规范和www.amqp.org站点上的其他资料，或者可以阅读Alvaro Videla和Jason J.W. Williams所编写的《RabbitMQ in Action》（Manning, 2012, www.manning.com/videla/）。

现在，我们结束对AMQP的抽象讨论，开始着手编写借助Spring AMQP发送和接收消息的代码。首先我们将看到的是一些通用的配置，它们同时适用于生产者和消费者。

17.3.2 配置Spring支持AMQP消息

当我们第一次使用Spring JMS抽象的时候，首先配置了一个连接工厂。与之类似，使用Spring AMQP前也要配置一个连接工厂。只不过，所要配置的不是JMS的连接工厂，而是需要配置AMQP的连接工厂。更具体来讲，需要配置RabbitMQ连接工厂。

什么是**RabbitMQ**

RabbitMQ是一个流行的开源消息代理，它实现了AMQP。Spring AMQP为RabbitMQ提供了支持，包括RabbitMQ连接工厂、模板以及Spring配置命名空间。

在使用它发送和接收消息之前，你需要预先安装RabbitMQ。我们可以在www.rabbitmq.com/download.html上找到安装指南。根据你所运行的OS不同，这会有所差别，所以根据环境的不同，遵循相应指南进行安装的任务就留给读者自己完成。

配置RabbitMQ连接工厂最简单的方式就是使用Spring AMQP所提供的`rabbit`配置命名空间。为了使用这项功能，需要确保在Spring配置文件中已经声明了该模式：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/rabbit"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/rabbit
   http://www.springframework.org/schema/rabbit/spring-rabbit-1.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

...

</beans:beans>
```

尽管不是必须的，但我还选择在这个配置中将rabbit作为首选的命名空间，将beans作为第二位的命名空间。这是因为在这个配置中，我会更多的声明rabbit而不是bean，这样的话，只会有少量的bean元素使用“beans:”前缀，而rabbit元素就能够避免使用前缀了。

rabbit命名空间包含了多个在Spring中配置RabbitMQ的元素。但此时，你最感兴趣的可能就是<connection-factory>。按照其最简单的形式，我们可以在配置RabbitMQ连接工厂的时候没有任何属性：

```
<connection-factory/>
```

这的确能够运行起来，但是所导致的结果就是连接工厂bean没有可用的bean ID，这样的话就难将连接工厂装配到需要它的bean中。因此，我们可能希望通过id属性为其设置一个bean ID：

```
<connection-factory id="connectionFactory" />
```

默认情况下，连接工厂会假设RabbitMQ服务器监听localhost的5672端口，并且用户名和密码均为guest。对于开发来讲，这是合理的默认值，但是对于生产环境，我们可能希望修改这些默认值。如下<connection-factory>的设置重写了默认的做法：

```
<connection-factory id="connectionFactory"
  host="${rabbitmq.host}"
  port="${rabbitmq.port}"
  username="${rabbitmq.username}"
  password="${rabbitmq.password}" />
```

我们使用占位符来指定值，这样配置项可以在Spring配置文件之外进行管理（很可能位于属性文件中）。

除了连接工厂以外，我们还要考虑使用其他的几个配置元素。接下来，看一下如何创建队列、Exchange以及binding。

声明队列、Exchange以及binding

在JMS中，队列和主题的路由行为都是通过规范建立的，AMQP与之不同，它的路由更加丰富和灵活，依赖于如何定义队列和Exchange以及如何将它们绑定在一起。声明队列、Exchange和binding的一种方式是使用RabbitMQ `Channel`接口的各种方法。但是直接使用RabbitMQ的`Channel`接口非常麻烦。Spring AMQP能否帮助我们声明消息路由组件呢？

幸好，rabbit命名空间包含了多个元素，帮助我们声明队列、Exchange以及将它们结合在一起的binding。表17.3中列出了这些元素。

表17.3　Spring AMQP的rabbit命名空间包含了多个元素，用来创建队列、Exchange以及将它们结合在一起的binding

元　　素	作　　用
`<queue>`	创建一个队列
`<fanout-exchange>`	创建一个fanout类型的Exchange
`<header-exchange>`	创建一个header类型的Exchange
`<topic-exchange>`	创建一个topic类型的Exchange
`<direct-exchange>`	创建一个direct类型的Exchange
`<bindings><binding/></bindings>`	元素定义一个或多个元素的集合。元素创建Exchange和队列之间的binding

这些配置元素要与`<admin>`元素一起使用。`<admin>`元素会创建一个RabbitMQ管理组件（administrative component），它会自动创建（如果它们在RabbitMQ代理中尚未存在的话）上述这些元素所声明的队列、Exchange以及binding。

例如，如果你希望声明名为`spittle.alert.queue`的队列，只需要在Spring配置中添加如下的两个元素即可：

```
<admin connection-factory="connectionFactory"/
     > <queue id="spittleAlertQueue" name="spittle.alerts" />
```

对于简单的消息来说，我们只需做这些就足够了。这是因为默认会有一个没有名称的direct Exchange，所有的队列都会绑定到这个Exchange上，并且routing key与队列的名称相同。在这个简单的配置中，我们可以将消息发送到这个没有名称的Exchange上，并将routing key设定为`spittle.alert.queue`，这样消息就会路由到这个队列中。实际上，我们重新创建了JMS的点对点模型。

但是，更加有意思的路由需要我们声明一个或更多的Exchange，并将其绑定到队列上。例如，如果要将消息路由到多个队列中，而不管routing key是什么，我们可以按照如下的方式配置一个fanout以及多个队列：

```
<admin connection-factory="connectionFactory" /
     > <queue name="spittle.alert.queue.1" > <queue name="spittle.alert.queue
     .2" > <queue name="spittle.alert.queue.3" > <fanout-
     exchange name="spittle.fanout">  <bindings>   <binding queue="spittle.al
     ert.queue.1" />   <binding queue="spittle.alert.queue.2" /
     >   <binding queue="spittle.alert.queue.3" />  </bindings> </fanout-
     exchange>
```

借助表17.3中的元素，会有无数种在RabbitMQ配置路由的方式，但是我却没有无尽的篇幅来为读者描述它们，所以为了让我们的讨论不至于偏离方向，我将这些创造性的路由作为练习留给读者，我将会继续讨论如何发送消息。

17.3.3 使用RabbitTemplate发送消息

顾名思义，RabbitMQ连接工厂的作用是创建到RabbitMQ的连接。如果你希望通过RabbitMQ发送消息，那么你可以将`connectionFactory`bean注入到`AlertServiceImpl`类中，并使用它来创建`Connection`，使用这个`Connection`来创建`Channel`，然后使用这个`Channel`发布消息到Exchange上。

是的，你的确可以这样做。

但是，如果这样做的话，你要做许多的工作并且会涉及到很多样板式代码。Spring所讨厌的一件事情就是样板式代码。我们已经看到Spring提供模板来消除样板式代码的多个例子——包括本章前面所介绍的`JmsTemplate`，它消除了JMS的样板式代码。因此，Spring AMQP提供`RabbitTemplate`来消除RabbitMQ发送和接收消息相关

的样板式代码就一点也不让人感觉奇怪了。

配置RabbitTemplate的最简单方式是使用rabbit命名空间的<template>元素，如下所示：

```
<template id="rabbitTemplate"
        connection-factory="connectionFactory" />
```

现在，要发送消息的话，我们只需要将模板bean注入到AlertServiceImpl中，并使用它来发送Spittle。如下的程序清单展现了一个新版本的AlertServiceImpl，它使用RabbitTemplate代替JmsTemplate来发送Spittle提醒。

程序清单**17.7**　使用**RabbitTemplate**来发送**Spittle**

```
package com.habuma.spitter.alerts;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;

import com.habuma.spitter.domain.Spittle;

public class AlertServiceImpl implements AlertService {

  private RabbitTemplate rabbit;

  @Autowired
  public AlertServiceImpl(RabbitTemplate rabbit) {
    this.rabbit = rabbit;
  }

  public void sendSpittleAlert(Spittle spittle) {
    rabbit.convertAndSend("spittle.alert.exchange",
                          "spittle.alerts",
                          spittle);
  }

}
```

可以看到，现在sendSpittleAlert()调用RabbitTemplate的convertAndSend()方法，其中RabbitTemplate是被注入进来的。它传入了三个参数：Exchange的名称、routing key以及要发送的对象。注意，这里并没有指定消息该路由到何处、要发送给哪个队列以及期望哪个消费者来获取消息。

RabbitTemplate有多个重载版本的convertAndSend()方法，这些方法可以简化它的使用。例如，使用某个重载版本的convertAndSend()方法，我们可以在调用convertAndSend()的时候，不设置Exchange的名称：

```
rabbit.convertAndSend("spittle.alerts", spittle);
```

如果你愿意的话，还可以同时省略Exchange名称和routing key：

```
rabbit.convertAndSend(spittle);
```

如果在参数列表中省略Exchange名称，或者同时省略Exchange名称和routing key的话，RabbitTemplate将会使用默认的Exchange名称和routing key。按照我们之前的配置，默认的Exchange名称为空（或者说是默认没有名称的那一个Exchange），默认的routing key也为空。但是，我们可以在<template>元素上借助exchange和routing-key属性配置不同的默认值：

```
<template id="rabbitTemplate"
connection-factory="connectionFactory"
exchange="spittle.alert.exchange"
routing-key="spittle.alerts" />
```

不管设置的默认值是什么，我们都可以在调用convertAndSend()方法的时候，以参数的形式显式指定它们，从而覆盖掉默认值。

RabbitTemplate还有其他的方法来发送消息，你可能会对此感兴趣。例如，我们可以使用较低等级的send()方法来发送org.springframework.amqp.core.Message对象，如下所示：

```
Message helloMessage =
    new Message("Hello World!".getBytes(), new MessageProperties());
rabbit.send("hello.exchange", "hello.routing", helloMessage);
```

与convertAndSend()方法类似，send()方法也有重载形式，它们不需要提供Exchange名称和/或routing key。

使用send()方法的技巧在于构造要发送的Message对象。在这个“Hello World”样例中，我们通过给定字符串的字节数组来构建Message实例。对于String值来说，这足够了，但是如果消息的负载是复杂对象的话，那它就会复杂得多。

鉴于这种情况，我们有了convertAndSend()方法，它会自动将对象转换为Message。它需要一个消息转换器的帮助来完成该任务，默认的消息转换器是SimpleMessageConverter，它适用于String、Serializable实例以及字节数组。Spring AMQP还提供了其他几个有用的消息转换器，其中包括使用JSON和XML数据的消息转换器。

现在，我们已经发送了消息，接下来我们转向回话的另外一端，看一

下如何获取消息。

17.3.4 接收AMQP消息

我们可以回忆一下，JMS提供了两种从队列中获取信息的方式：使用JmsTemplate的同步方式以及使用消息驱动POJO的异步方式。Spring AMQP提供了类似的方式来获取通过AMQP发送的消息。因为我们已经有了RabbitTemplate，所以首先看一下如何使用它同步地从队列中获取消息。

使用RabbitTemplate来接收消息

RabbitTemplate提供了多个接收信息的方法。最简单就是receive()方法，它位于消息的消费者端，对应于RabbitTemplate的send()方法。借助receive()方法，我们可以从队列中获取一个Message对象：

```
Message message = rabbit.receive("spittle.alert.queue");
```

或者，如果愿意的话，你还可以配置获取消息的默认队列，这是通过在配置模板的时候，设置queue属性实现的：

```
<template id="rabbitTemplate"
    connection-factory="connectionFactory"
    exchange="spittle.alert.exchange"
    routing-key="spittle.alerts"
    queue="spittle.alert.queue" />
```

这样的话，我们在调用receive()方法的时候，不需要设置任何参数就能从默认队列中获取消息了：

```
Message message = rabbit.receive();
```

在获取到Message对象之后，我们可能需要将它body属性中的字节数组转换为想要的对象。就像在发送的时候将领域对象转换为Message一样，将接收到的Message转换为领域对象同样非常繁琐。因此，我们可以考虑使用RabbitTemplate的receiveAndConvert()方法作为替代方案：

```
Spittle spittle =
    (Spittle) rabbit.receiveAndConvert("spittle.alert.queue");
```

我们还可以省略调用参数中的队列名称，这样它就会使用模板的默认队列名称：

```
Spittle spittle = (Spittle) rabbit.receiveAndConvert();
```

receiveAndConvert()方法会使用与sendAndConvert()方法相同的消息转换器，将Message对象转换为原始的类型。

调用receive()和receiveAndConvert()方法都会立即返回，如果队列中没有等待的消息时，将会得到null。这就需要我们来管理轮询（polling）以及必要的线程，实现队列的监控。

我们并非必须同步轮询并等待消息到达，Spring AMQP还提供了消息驱动POJO的支持，这不禁使我们回忆起Spring JMS中的相同特性。让我们看一下如何通过消息驱动AMQP POJO的方式来接收消息。

定义消息驱动的**AMQP POJO**

如果你想在消息驱动POJO中异步地消费使用Spittle对象，首先要解决的问题就是这个POJO本身。如下的SpittleAlertHandler扮演了这个角色：

```
package com.habuma.spittr.alerts;
import com.habuma.spittr.domain.Spittle;

public class SpittleAlertHandler {

  public void handleSpittleAlert(Spittle spittle) {
    // ... implementation goes here ...
  }

}
```

注意，这个类与借助JMS消费Spittle时所用到SpittleAlertHandler完全一致。我们之所以能够重用相同的POJO是因为这个类丝毫没有依赖于JMS或AMQP，并且不管通过什么机制传递过来Spittle对象，它都能够进行处理。

我们还需要在Spring应用上下文中将SpittleAlertHandler声明为一个bean：

```
<bean id="spittleListener"
      class="com.habuma.spittr.alert.SpittleAlertHandler" />
```

同样，在使用基于JMS的MDP时，我们已经做过相同的事情，没有什么丝毫的差异。

最后，我们需要声明一个监听器容器和监听器，当消息到达的时候，能够调用SpittleAlertHandler。在基于JMS的MDP中，我们做过相同的事情，但是基于AMQP的MDP在配置上有一个细微的差别：

```
<listener-container connection-factory="connectionFactory">
    <listener ref="spittleListener"
          method="handleSpittleAlert"
          queue-names="spittle.alert.queue" />
</listener-container>
```

你看到有什么差别了吗？我也同意这并不那么明显。<listener-container>与<listener>都与JMS对应的元素非常类似。但是，这些元素来自rabbit命名空间，而不是JMS命名空间。

我都说过了，没那么明显。

哦，还有一个细微的差别，我们不再通过destination属性（JMS中的做法）来监听队列或主题，这里我们通过queue-names属性来指定要监听的队列。但是，除此之外，基于AMQP的MDP与基于JMS的MDP都非常类似。

你可能也意识到了，queue-names属性的名称使用了复数形式。在这里我们只设定了一个要监听的队列，但是允许设置多个队列的名称，用逗号分割即可。

另外一种指定要监听队列的方法是引用<queue>元素所声明的队列bean。我们可以通过queues属性来进行设置：

```
<listener-container connection-factory="connectionFactory">
    <listener ref="spittleListener"
          method="handleSpittleAlert"
          queues="spittleAlertQueue" />
</listener-container>
```

同样，这里可以接受逗号分割的queue ID列表。当然，这需要我们在声明队列的时候，为其指定ID。例如，如下是重新定义的提醒队列，这次指定了ID：

```
<queue id="spittleAlertQueue" name="spittle.alert.queue" />
```

注意，这里的id属性用来在Spring应用上下文中设置队列的bean ID，而name属性指定了RabbitMQ代理中队列的名称。

17.4 小结

异步消息通信与同步RPC相比有几个优点。间接通信带来了应用之间的松散耦合，因此减轻了其中任意一个应用崩溃所带来的影响。此外，因为消息转发给了收件人，因此发送者不必等待响应。在很多情

况下，这会提高应用的性能。

虽然JMS为所有的Java应用程序提供了异步通信的标准API，但是它使用起来很繁琐。Spring消除了JMS样板式代码和异常捕获代码，让异步消息通信更易于使用。

在本章中，我们了解了Spring通过消息代理和JMS建立应用程序之间异步通信的几种方式。Spring的JMS模板消除了传统的JMS编程模型所必需的样板式代码，而基于Spring的消息驱动bean可以通过声明bean的方法允许方法响应来自于队列或主题中的消息。我们同样了解了如何通过Spring的JMS invoker为Spring bean提供基于消息的RPC。

在本章中，我们已经看到了如何在应用程序之间使用异步通信。在下一章中，我们将会延续这一话题，了解如何借助WebSocket在基于浏览器的客户端和服务器之间实现异步通信。

---

[1]作者幽默夸张的说法。——译者注

[2]如果读到此处，你觉得AMQP能够不局限于Java语言和平台，那说明你已经快速抓到了重点。

[3]有一点我还没有提到，那就是可以将某个Exchange绑定到另外一个Exchange上，创建路由的内嵌等级结构。

第18章　使用WebSocket和STOMP实现消息功能

本章内容：

- 在浏览器和服务器之间发送消息
- 在Spring MVC控制器中处理消息
- 为目标用户发送消息

在上一章中，我们看到了如何使用JMS和AMQP在应用程序之间发送消息。异步消息是应用程序之间通用的交流方式。但是，如果某一应用是运行在Web浏览器中，那我们就需要一些稍微不同的技巧了。

WebSocket协议提供了通过一个套接字实现全双工通信的功能。除了其他的功能之外，它能够实现Web浏览器和服务器之间的异步通信。全双工意味着服务器可以发送消息给浏览器，浏览器也可以发送消息给服务器。

Spring 4.0为WebSocket通信提供了支持，包括：

- 发送和接收消息的低层级API；
- 发送和接收消息的高级API；
- 用来发送消息的模板；
- 支持SockJS，用来解决浏览器端、服务器以及代理不支持WebSocket的问题。

在本章中，我们将会学习借助Spring的WebSocket功能实现服务器端和基于浏览器的应用之间实现异步通信。我们首先会从如何使用Spring的低层级WebSocket API开始。

18.1　使用Spring的低层级WebSocket API

按照其最简单的形式，WebSocket只是两个应用之间通信的通道。位于WebSocket一端的应用发送消息，另外一端处理消息。因为它是全双工的，所以每一端都可以发送和处理消息。如图18.1所示。

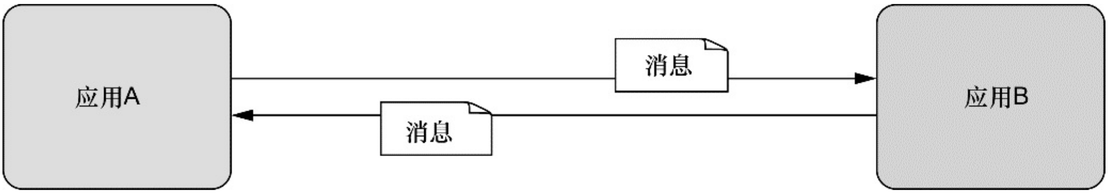

图18.1 WebSocket是两个应用之间全双工的通信通道

WebSocket通信可以应用于任何类型的应用中，但是WebSocket最常见的应用场景是实现服务器和基于浏览器的应用之间的通信。浏览器中的JavaScript客户端开启一个到服务器的连接，服务器通过这个连接发送更新给浏览器。相比历史上轮询服务端以查找更新的方案，这种技术更加高效和自然。

为了阐述Spring低层级的WebSocket API，让我们编写一个简单的WebSocket样例，基于JavaScript的客户端与服务器玩一个无休止的“Marco Polo”游戏。服务器端的应用会处理文本消息（“Marco!”），然后在相同的连接上往回发送文本消息（“Polo!”）。为了在Spring使用较低层级的API来处理消息，我们必须编写一个实现`WebSocketHandler`的类：

```
public interface WebSocketHandler {
  void afterConnectionEstablished(WebSocketSession session)
                                                        throws Exception;
  void handleMessage(WebSocketSession session,
                     WebSocketMessage<?> message) throws Exception;
  void handleTransportError(WebSocketSession session,
                            Throwable exception) throws Exception;
  void afterConnectionClosed(WebSocketSession session,
                             CloseStatus closeStatus) throws Exception;
  boolean supportsPartialMessages();
}
```

可以看到，`WebSocketHandler`需要我们实现五个方法。相比直接实现`WebSocketHandler`，更为简单的方法是扩展`AbstractWebSocketHandler`，这是`WebSocketHandler`的一个抽象实现。如下的程序清单展现了`MarcoHandler`，它是`AbstractWebSocketHandler`的一个子类，会在服务器端处理消息。

程序清单**18.1** **MarcoHandler**处理通过**WebSocket**传送的文本消息

```
package marcopolo;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler.AbstractWebSocketHandler;

public class MarcoHandler extends AbstractWebSocketHandler {

  private static final Logger logger =
      LoggerFactory.getLogger(MarcoHandler.class);          // 处理文本消息

  protected void handleTextMessage(
       WebSocketSession session, TextMessage message) throws Exception {
    logger.info("Received message: " + message.getPayload());

    Thread.sleep(2000);                                      // 模拟延时

    session.sendMessage(new TextMessage("Polo!"));           // 发送文本消息
   }

}
```

尽管AbstractWebSocketHandler是一个抽象类，但是它并不要求我们必须重载任何特定的方法。相反，它让我们来决定该重载哪一个方法。除了重载WebSocketHandler中所定义的五个方法以外，我们还可以重载AbstractWebSocketHandler中所定义的三个方法：

- handleBinaryMessage()
- handlePongMessage()
- handleTextMessage()

这三个方法只是handleMessage()方法的具体化，每个方法对应于某一种特定类型的消息。

因为MarcoHandler将会处理文本类型的“Marco!”消息，因此我们应该重载handleTextMessage()方法。当有文本消息抵达的时候，日志会记录消息内容，在两秒钟的模拟延迟之后，在同一个连接上返回另外一条文本消息。

MarcoHandler所没有重载的方法都由AbstractWebSocketHandler以空操作的方式（no-op）进行了实现。这意味着MarcoHandler也能处理二进制和pong消息，只是对这些消息不进行任何操作而已。

另外一种方案，我们可以扩展TextWebSocketHandler，不再扩展Abstract-WebSocketHandler：

```
public class MarcoHandler extends TextWebSocketHandler {
  ...
}
```

TextWebSocketHandler是AbstractWebSocketHandler的子类，它会拒绝处理二进制消息。它重载了handleBinaryMessage()方法，如果收到二进制消息的时候，将会关闭WebSocket连接。与之类似，BinaryWebSocketHandler也是AbstractWeb-SocketHandler的子类，它重载了handleTextMessage()方法，如果接收到文本消息的话，将会关闭连接。

尽管你会关心如何处理文本消息或二进制消息，或者二者兼而有之，但是你可能还会对建立和关闭连接感兴趣。在本例中，我们可以重载afterConnectionEstablished()和afterConnectionClosed()：

```
public void afterConnectionEstablished(WebSocketSession session)
    throws Exception {
  logger.info("Connection established");
}

@Override
public void afterConnectionClosed(
    WebSocketSession session, CloseStatus status) throws Exception {
  logger.info("Connection closed. Status: " + status);
}
```

我们通过afterConnectionEstablished()和afterConnectionClosed()方法记录了连接信息。当新连接建立的时候，会调用afterConnectionEstablished()方法，类似地，当连接关闭时，会调用afterConnectionClosed()方法。在本例中，连接事件仅仅记录了日志，但是如果我们想在连接的生命周期上建立或销毁资源时，这些方法会很有用。

注意，这些方法都是以“after”开头。这意味着，这些事件只能在事件发生后才产生响应，因此并不能改变结果。

现在，已经有了消息处理器类，我们必须要对其进行配置，这样Spring才能将消息转发给它。在Spring的Java配置中，这需要在一个配置类上使用@EnableWebSocket，并实现WebSocketConfigurer接口，如下面的程序清单所示。

程序清单**18.2**　在**Java**配置中，启用**WebSocket**并映射消息处理器

```
package marcopolo;

import org.springframework.context.annotation.Bean;
import org.springframework.web.socket.config.annotation.
                                            EnableWebSocket;
import org.springframework.web.socket.config.annotation.
                                            WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.
                                            WebSocketHandlerRegistry;
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

  @Override
  public void registerWebSocketHandlers(
                               WebSocketHandlerRegistry registry) {

    registry.addHandler(marcoHandler(), "/marco");   <-- 将 MarcoHandler 映射到“/marco”
   }

  @Bean
  public MarcoHandler marcoHandler() {   <-- 声明 MarcoHandler bean
     return new MarcoHandler();
  }

}
```

registerWebSocketHandlers()方法是注册消息处理器的关键。通过重载该方法，我们得到了一个WebSocketHandlerRegistry对象，通过该对象可以调用addHandler()来注册信息处理器。在本例中，我们注册了MarcoHandler（以bean的方式进行声明）并将其与“/marco”路径相关联。

另外，如果你更喜欢使用XML来配置Spring的话，那么可以使用websocket命名空间：

程序清单18.3　借助websocket命名空间以XML的方式配置WebSocket

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:websocket="http://www.springframework.org/schema/websocket"
 xsi:schemaLocation="
  http://www.springframework.org/schema/websocket
  http://www.springframework.org/schema/websocket/spring-websocket.xsd
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <websocket:handlers>
    <websocket:mapping handler="marcoHandler" path="/marco" />   <-- 将 Marco Handle 映射到“/marco”
   </websocket:handlers>

  <bean id="marcoHandler"
        class="marcopolo.MarcoHandler" />   <-- 声明 MarcoHandler bean

</beans>
```

不管使用Java还是使用XML，这就是所需的配置。

现在，我们可以把注意力转向客户端，它会发送“Marco!”文本消息到服务器，并监听来自服务器的文本消息。如下程序清单所展示的JavaScript代码开启了一个原始的WebSocket并使用它来发送消息给服务器。

程序清单**18.4**　连接到**“marco” WebSocket**的**JavaScript**客户端

```
var url = 'ws://' + window.location.host + '/websocket/marco';
var sock = new WebSocket(url);                          ◁—— 打开 WebSocket

sock.onopen = function() {                              ◁—— 处理连接开启事件
   console.log('Opening');
  sayMarco();
};

sock.onmessage = function(e) {                          ◁—— 处理信息
   console.log('Received message: ', e.data);
  setTimeout(function(){sayMarco()}, 2000);
};

sock.onclose = function() {                             ◁—— 处理连接关闭事件
   console.log('Closing');
};

function sayMarco() {
  console.log('Sending Marco!');
  sock.send("Marco!");                                  ◁—— 发送消息
 }
```

在程序清单18.4的代码中，所做的第一件事情就是创建`WebSocket`实例。对于支持WebSocket的浏览器来说，这个类型是原生的。通过创建`WebSocket`实例，实际上打开了到给定URL的WebSocket。在本例中，URL使用了“ws://”前缀，表明这是一个基本的WebSocket连接。如果是安全WebSocket的话，协议的前缀将会是“wss://”。

`WebSocket`创建完毕之后，接下来的代码建立了`WebSocket`的事件处理功能。注意，`WebSocket`的`onopen`、`onmessage`和`onclose`事件对应于`MarcoHandler`的`after-ConnectionEstablished()`、`handleTextMessage()`和`afterConnectionClosed()`方法。在`onopen`事件中，设置了一个函数，它会调用`sayMarco()`方法，在该WebSocket上发送“Marco!”消息。通过发送“Marco!”，这个无休止的Marco Polo游戏就开始了，因为服务器端的`MarcoHandler`作为响应会将“Polo!”发送回来，当客户端收到来自服务器的消息后，`onmessage`事件会发送另外一个“Marco!”给服务器。

这个过程会一直持续下去，直到连接关闭。在程序清单18.4中所没有展示的是如果调用`sock.close()`的话，将会结束这个疯狂的游戏。在服务端也可以关闭连接，或者浏览器转向其他的页面，都会关闭连接。如果发生以上任意的场景，只要连接关闭，都会触发onclose

事件。在这里，出现这种情况将会在控制台日志上记录一条信息。

到此为止，我们已经编写完使用Spring低层级WebSocket API的所有代码，包括接收和发送消息的处理器类，以及在浏览器端完成相同功能的JavaScript客户端。如果我们构建这些代码并将其部署到Servlet容器中，那它有可能能够正常运行。

从我选择“可能”这个词，你是不是能够感觉到这里有一点悲观的情绪？这是因为我不能保证它可以正常运行。实际上，它很有可能运行不起来。即便把所有的事情都做对了，诡异的事情依然会困扰我们。

让我们看一下都有什么事情会阻止WebSocket正常运行，并采取一些措施提高成功的几率。

18.2 应对不支持WebSocket的场景

WebSocket是一个相对比较新的规范。虽然它早在2011年底就实现了规范化，但即便如此，在Web浏览器和应用服务器上依然没有得到一致的支持。Firefox和Chrome早就已经完整支持WebSocket了，但是其他的一些浏览器刚刚开始支持WebSocket。如下列出了几个流行的浏览器支持WebSocket功能的最低版本：

- Internet Explorer：10.0
- Firefox: 4.0（部分支持），6.0（完整支持）。
- Chrome: 4.0（部分支持），13.0（完整支持）。
- Safari: 5.0（部分支持），6.0（完整支持）。
- Opera: 11.0（部分支持），12.10（完整支持）。
- iOS Safari: 4.2（部分支持），6.0（完整支持）。
- Android Browser: 4.4。

令人遗憾的是，很多的网上冲浪者并没有认识到或理解新Web浏览器的特性，因此升级很慢。另外，有的公司规定使用特定版本的浏览器，这样它们的员工很难（或不可能）使用更新的浏览器。鉴于这些情况，如果你的应用程序使用WebSocket的话，用户可能会无法使用。

服务器端对WebSocket的支持也好不到哪里去。GlassFish在几年前就开始支持一定形式的WebSocket，但是很多其他的应用服务器在最近的版本中刚刚开始支持WebSocket。例如，我在测试上述例子的时候，所使用的就是Tomcat 8的发布候选构建版本。

即便浏览器和应用服务器的版本都符合要求，两端都支持WebSocket，在这两者之间还有可能出现问题。防火墙代理通常会限制所有除HTTP以外的流量。它们有可能不支持或者（还）没有配置允许进行WebSocket通信。

在当前的WebSocket领域，我也许描述了一个很阴暗的前景。但是，不要因为这一些不支持，你就停止使用WebSocket的功能。当它能够正常使用的时候，WebSocket是一项非常棒的技术，但是如果它无法得到支持的话，我们所需要的仅仅是一种备用方案（fallback plan）。

幸好，提到WebSocket的备用方案，这恰是SockJS所擅长的。SockJS是WebSocket技术的一种模拟，在表面上，它尽可能对应WebSocket API，但是在底层它非常智能，如果WebSocket技术不可用的话，就会选择另外的通信方式。SockJS会优先选用WebSocket，但是如果WebSocket不可用的话，它将会从如下的方案中挑选最优的可行方案：

- XHR流。
- XDR流。
- iFrame事件源。
- iFrame HTML文件。
- XHR轮询。
- XDR轮询。
- iFrame XHR轮询。
- JSONP轮询。

好消息是在使用SockJS之前，我们并没有必要全部了解这些方案。SockJS让我们能够使用统一的编程模型，就好像在各个层面都完整支持WebSocket一样，SockJS在底层会提供备用方案。

例如，为了在服务端启用SockJS通信，我们在Spring配置中可以很简单地要求添加该功能。重新回顾一下程序清单18.2中的`registerWebSocketHandlers()`方法，稍微加一点内容就能启用SockJS：

```
@Override
public void registerWebSocketHandlers(
                                WebSocketHandlerRegistry registry) {
  registry.addHandler(marcoHandler(), "/marco").withSockJS();
}
```

`addHandler()`方法会返回`WebSocketHandlerRegistration`，

通过简单地调用其withSockJS()方法就能声明我们想要使用SockJS功能，如果WebSocket不可用的话，SockJS的备用方案就会发挥作用。

如果你使用XML来配置Spring的话，启用SockJS只需在配置中添加<websocket:sockjs>元素即可：

```
<websocket:handlers>
  <websocket:mapping handler="marcoHandler" path="/marco" />
  <websocket:sockjs />
</websocket:handlers>
```

要在客户端使用SockJS，需要确保加载了SockJS客户端库。具体的做法在很大程度上依赖于使用JavaScript模块加载器（如require.js或curl.js）还是简单地使用<script>标签加载JavaScript库。加载SockJS客户端库的最简单办法是使用<script>标签从SockJS CDN中进行加载，如下所示：

```
<script src="http://cdn.sockjs.org/sockjs-0.3.min.js"></script>
```

用**WebJars**解析**Web**资源

在我的样例代码中，使用了WebJars来解析JavaScript库，使其作为项目Maven或Gradle构建的一部分，就像其他的依赖一样。为了支持该功能，我在Spring MVC配置中搭建了一个资源处理器，让它负责解析路径以“/webjars/**”开头的请求，这也是WebJars的标准路径：

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
  registry.addResourceHandler("/webjars/**")
      .addResourceLocations("classpath:/META-INF/resources/webjars/");
}
```

在这个资源处理器准备就绪后，我们可以在Web页面中使用如下的<script>标签加载SockJS库：

```
<script th:src="@{/webjars/sockjs-client/0.3.4/sockjs.min.js}">
</script>
```

注意，这个特殊的<script>标签来源于一个Thymeleaf模板，并使用“@{...}”表达式来为JavaScript文件计算完整的相对于上下文的URL路径。

除了加载SockJS客户端库以外，在程序清单18.4中，要使用SockJS只需修改两行代码：

```
var url = 'marco';
var sock = new SockJS(url);
```

所做的第一个修改就是URL。SockJS所处理的URL是“http://”或“https://”模式，而不是“ws://”和“wss://”。即便如此，我们还是可以使用相对URL，避免书写完整的全限定URL。在本例中，如果包含JavaScript的页面位于“http://localhost:8080/websocket”路径下，那么给定的“marco”路径将会形成到“http://localhost:8080/websocket/marco”的连接。

但是，这里最核心的变化是创建SockJS实例来代替WebSocket。因为SockJS尽可能地模拟了WebSocket，所以程序清单18.4中的其他代码并不需要变化。相同的onopen、onmessage和onclose事件处理函数用来响应对应的事件，相同的send()方法用来发送“Marco!”到服务器端。

我们并没有改变很多的代码，但是客户端-服务器之间通信的运行方式却有了很大的变化。我们可以完全相信客户端和服务器之间能够进行类似于WebSocket这样的通信，即便浏览器、服务器或位于中间的代理不支持WebSocket，我们也无需再担心了。

WebSocket提供了浏览器-服务器之间的通信方式，当运行环境不支持WebSocket的时候，SockJS提供了备用方案。但是不管哪种场景，对于实际应用来说，这种通信形式都显得层级过低。让我们看一下如何在WebSocket之上使用STOMP（Simple Text Oriented Messaging Protocol），为浏览器-服务器之间的通信增加恰当的消息语义。

18.3　使用STOMP消息

如果我要求你编写一个Web应用程序，在讨论需求之前，你可能对于要采用的基础技术和框架就有了很好的想法。即便是简单的“Hello World”Web应用，你可能也会考虑使用Spring MVC控制器来处理请求，并为响应使用JSP或Thymeleaf模板。至少，你也应该会创建一个静态的HTML页面，并让Web服务器处理来自Web浏览器的相应请求。我们应该不会关心浏览器具体如何请求页面以及页面如何传递给浏览器这样的事情。

现在，我们假设HTTP协议并不存在，只能使用TCP套接字来编写Web应用。你可能认为我已经疯掉了。当然，我们也许能够完成这一壮举，但是这需要自行设计客户端和服务器端都认可的协议，从而实现

有效的通信。简单来说，这不是一件容易的事情。

不过，幸好我们有HTTP，它解决了Web浏览器发起请求以及Web服务器响应请求的细节。这样的话，大多数的开发人员并不需要编写低层级TCP套接字通信相关的代码。

直接使用WebSocket（或SockJS）就很类似于使用TCP套接字来编写Web应用。因为没有高层级的线路协议（wire protocol），因此就需要我们定义应用之间所发送消息的语义，还需要确保连接的两端都能遵循这些语义。

不过，好消息是我们并非必须要使用原生的WebSocket连接。就像HTTP在TCP套接字之上添加了请求-响应模型层一样，STOMP在WebSocket之上提供了一个基于帧的线路格式（frame-based wire format）层，用来定义消息的语义。

乍看上去，STOMP的消息格式非常类似于HTTP请求的结构。与HTTP请求和响应类似，STOMP帧由命令、一个或多个头信息以及负载所组成。例如，如下就是发送数据的一个STOMP帧：

```
SEND
destination:/app/marco
content-length:20

{\"message\":\"Marco!\"}
```

在这个简单的样例中，STOMP命令是`send`，表明会发送一些内容。紧接着是两个头信息：一个用来表示消息要发送到哪里的目的地，另外一个则包含了负载的大小。然后，紧接着是一个空行，STOMP帧的最后是负载内容，在本例中，是一个JSON消息。

STOMP帧中最有意思的恐怕就是`destination`头信息了。它表明STOMP是一个消息协议，类似于JMS或AMQP。消息会发布到某个目的地，这个目的地实际上可能真的有消息代理（message broker）作为支撑。另一方面，消息处理器（message handler）也可以监听这些目的地，接收所发送过来的消息。

在WebSocket通信中，基于浏览器的JavaScript应用可能会发送消息到一个目的地，这个目的地由服务器端的组件来进行处理。其实，反过来是一样的，服务器端的组件也可以发布消息，由JavaScript客户端的目的地来接收。

Spring为STOMP消息提供了基于Spring MVC的编程模型。稍后将会看

到，在Spring MVC控制器中处理STOMP消息与处理HTTP请求并没有太大的差别。但首先，我们需要配置Spring启用基于STOMP的消息。

18.3.1 启用STOMP消息功能

稍后，我们将会看到如何在Spring MVC中为控制器方法添加@MessageMapping注解，使其处理STOMP消息，它与带有@RequestMapping注解的方法处理HTTP请求的方式非常类似。但是与@RequestMapping不同的是，@MessageMapping的功能无法通过@EnableWebMvc启用。Spring的Web消息功能基于消息代理（message broker）构建，因此除了告诉Spring我们想要处理消息以外，还有其他的内容需要配置。我们必须要配置一个消息代理和其他的一些消息目的地。

如下的程序清单展现了如何通过Java配置启用基于代理的Web消息功能：

程序清单**18.5 @EnableWebSocketMessageBroker**注解能够在**WebSocket**之上启用**STOMP**

```
package marcopolo;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.config.annotation.
                              AbstractWebSocketMessageBrokerConfigurer;
import org.springframework.web.socket.config.annotation.
                                          EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.
                                                 StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker                        <—启用 STOMP 消息
 public class WebSocketStompConfig
       extends AbstractWebSocketMessageBrokerConfigurer {

  @Override
  public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/marcopolo").withSockJS();        <—为“/marcopolo”
   }                                                          路径启用 SockJS
                                                              功能
  @Override
  public void configureMessageBroker(MessageBrokerRegistry registry) {
    registry.enableSimpleBroker("/queue", "/topic");
    registry.setApplicationDestinationPrefixes("/app");
  }

}
```

与程序清单18.2中的配置进行对比，WebSocketStompConfig使用了@EnableWebSocketMessageBroker注解。这表明这个配置类不仅配置了WebSocket，还配置了基于代理的STOMP消息。它重载了registerStompEndpoints()方法，将“/marcopolo”注册为STOMP

端点。这个路径与之前发送和接收消息的目的地路径有所不同。这是一个端点，客户端在订阅或发布消息到目的地路径前，要连接该端点。

WebSocketStompConfig还通过重载configureMessageBroker()方法配置了一个简单的消息代理。这个方法是可选的，如果不重载它的话，将会自动配置一个简单的内存消息代理，用它来处理以“/topic”为前缀的消息。但是在本例中，我们重载了这个方法，所以消息代理将会处理前缀为“/topic”和“/queue”的消息。除此之外，发往应用程序的消息将会带有“/app”前缀。图18.2展现了这个配置中的消息流。

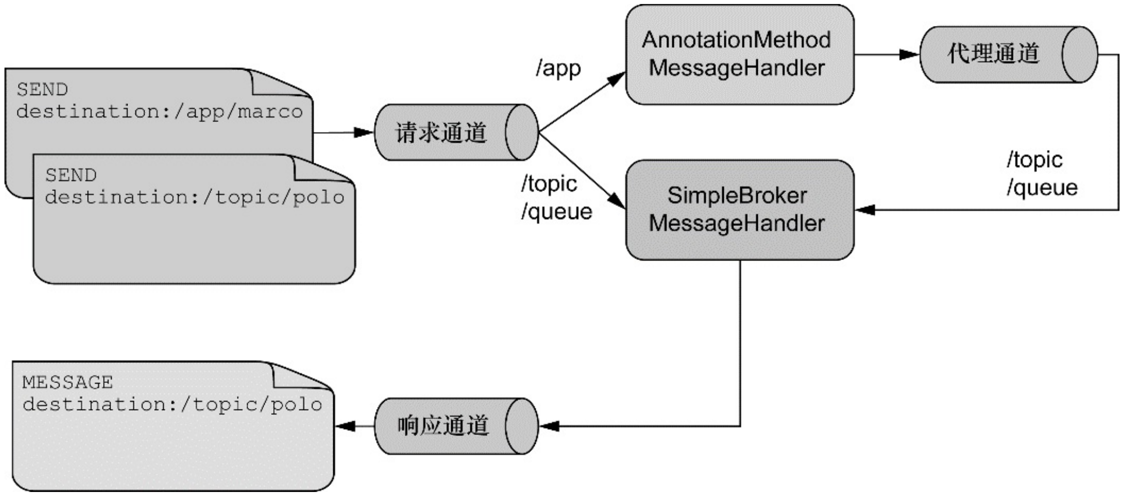

图18.2　Spring简单的STOMP代理是基于内存的，它模拟了STOMP代理的多项功能

当消息到达时，目的地的前缀将会决定消息该如何处理。在图18.2中，应用程序的目的地以“/app”作为前缀，而代理的目的地以“/topic”和“/queue”作为前缀。以应用程序为目的地的消息将会直接路由到带有@MessageMapping注解的控制器方法中。而发送到代理上的消息，其中也包括@MessageMapping注解方法的返回值所形成的消息，将会路由到代理上，并最终发送到订阅这些目的地的客户端。

启用**STOMP**代理中继

对于初学来讲，简单的代理是很不错的，但是它也有一些限制。尽管它模拟了STOMP消息代理，但是它只支持STOMP命令的子集。因为它是基于内存的，所以它并不适合集群，因为如果集群的话，每个节点也只能管理自己的代理和自己的那部分消息。

对于生产环境下的应用来说，你可能会希望使用真正支持STOMP的代理来支撑WebSocket消息，如RabbitMQ或ActiveMQ。这样的代理提供了可扩展性和健壮性更好的消息功能，当然它们也会完整支持STOMP命令。我们需要根据相关的文档来为STOMP搭建代理。搭建就绪之后，就可以使用STOMP代理来替换内存代理了，只需按照如下方式重载`configureMessageBroker()`方法即可：

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {
  registry.enableStompBrokerRelay("/topic", "/queue");
  registry.setApplicationDestinationPrefixes("/app");
}
```

上述`configureMessageBroker()`方法的第一行代码启用了STOMP代理中继（broker relay）功能，并将其目的地前缀设置为“/topic”和“/queue”。这样的话，Spring就能知道所有目的地前缀为“/topic”或“/queue”的消息都会发送到STOMP代理中。根据你所选择的STOMP代理不同，目的地的可选前缀也会有所限制。例如，RabbitMQ只允许目的地的类型为“/temp-queue”、“/exchange”、“/topic”、“/queue”、“/amq/queue”和“/reply-queue”。请参阅代理的文档来了解所支持的目的地类型及其使用场景。

除了目的地前缀，在第二行的`configureMessageBroker()`方法中将应用的前缀设置为“/app”。所有目的地以“/app”打头的消息都将会路由到带有`@MessageMapping`注解的方法中，而不会发布到代理队列或主题中。

图18.3阐述了代理中继如何应用于Spring的STOMP消息处理之中。我们可以看到，关键的区别在于这里不再模拟STOMP代理的功能，而是由代理中继将消息传送到一个真正的消息代理中来进行处理。

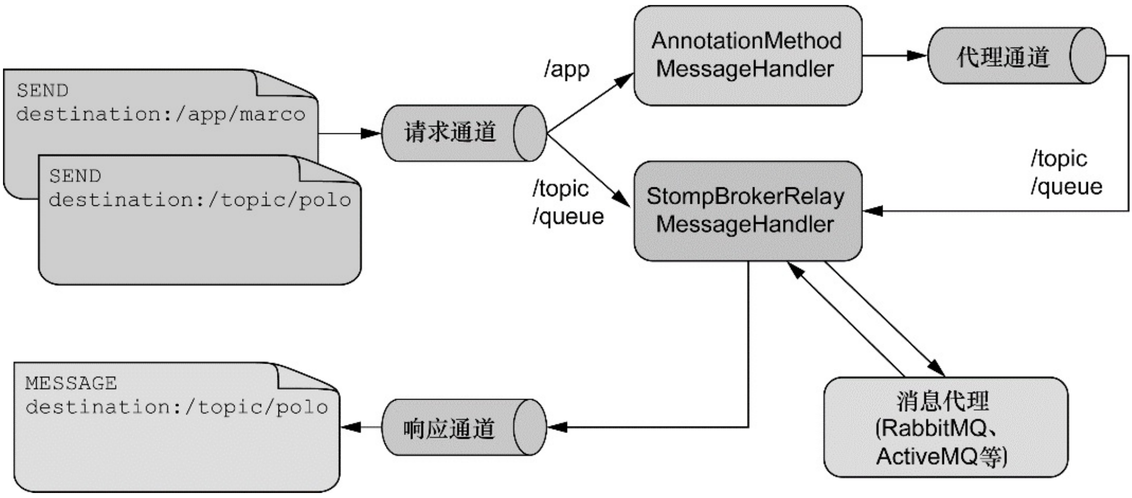

图18.3　STOMP代理中继会将STOMP消息的处理委托给一个真正的消息代理

注意，enableStompBrokerRelay()和setApplicationDestinationPrefixes()方法都接收可变长度的String参数，所以我们可以配置多个目的地和应用前缀。例如：

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {
  registry.enableStompBrokerRelay("/topic", "/queue");
  registry.setApplicationDestinationPrefixes("/app", "/foo");
}
```

默认情况下，STOMP代理中继会假设代理监听localhost的61613端口，并且客户端的username和password均为“guest”。如果你的STOMP代理位于其他的服务器上，或者配置成了不同的客户端凭证，那么我们可以在启用STOMP代理中继的时候，需要配置这些细节信息：

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {
  registry.enableStompBrokerRelay("/topic", "/queue")
          .setRelayHost("rabbit.someotherserver")
          .setRelayPort(62623)
          .setClientLogin("marcopolo")
          .setClientPasscode("letmein01");
  registry.setApplicationDestinationPrefixes("/app", "/foo");
}
```

以上的这个配置调整了服务器、端口以及凭证信息。但是，并不是必须要配置所有的这些选项。例如，如果你只想修改中继端口，那么可以只调用setRelayHost()方法，在配置中不必使用其他的Setter方法。

现在，Spring已经配置就绪，可以用来处理STOMP消息了。

18.3.2　处理来自客户端的STOMP消息

我们在第5章已经学习过，Spring MVC为处理HTTP Web请求提供了面向注解的编程模型。@RequestMapping是Spring MVC中最著名的注解，它会将HTTP请求映射到对请求进行处理的方法上。在第16章，我们也曾经看到相同的编程模型扩展到了RESTful的资源处理中。

STOMP和WebSocket更多的是关于异步消息，与HTTP的请求-响应方式有所不同。但是，Spring提供了非常类似于Spring MVC的编程模型来处理STOMP消息。它非常地相似，以至于对STOMP消息的处理器方法也会包含在带有@Controller注解的类中。

Spring 4.0引入了@MessageMapping注解，它用于STOMP消息的处理，类似于Spring MVC的@RequestMapping注解。当消息抵达某个特定的目的地时，带有@MessageMapping注解的方法能够处理这些消息。例如，考虑如下程序清单中的控制器类。

程序清单**18.6**　借助**@MessageMapping**注解能够在控制器中处理**STOMP**消息

```
package marcopolo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.stereotype.Controller;

@Controller
public class MarcoController {

  private static final Logger logger =
      LoggerFactory.getLogger(MarcoController.class);

  @MessageMapping("/marco")                                  <-- 处理发往"/app/marco"目的地的消息
   public void handleShout(Shout incoming) {
    logger.info("Received message: " + incoming.getMessage());
  }

}
```

乍一看上去，它非常类似于其他的Spring MVC控制器类。它使用了@Controller注解，所以组件扫描能够找到它并将其注册为bean。就像其他的@Controller类一样，它也包含了处理器方法。

但是这个处理器方法与我们之前看到的有一点区别。handleShout()方法没有使用@RequestMapping注解，而是使用了@MessageMapping注解。这表示handleShout()方法能够处理指定目的地上到达的消息。在本例中，这个目的地也就是“/app/marco”（“/app”前缀是隐含的，因为我们将其配置为应用的目的地前缀）。

因为handleShout()方法接收一个Shout参数，所以Spring的某一个消息转换器会将STOMP消息的负载转换为Shout对象。Shout类非常简单，它是只具有一个属性的JavaBean，包含了消息的内容：

```
package marcopolo;

public class Shout {
  private String message;

  public String getMessage() {
    return message;
  }

  public void setMessage(String message) {
    this.message = message;
  }
}
```

因为我们现在处理的不是HTTP，所以无法使用Spring的HttpMessageConverter实现将负载转换为Shout对象。Spring 4.0提供了几个消息转换器，作为其消息API的一部分。表18.1描述了这些消息转换器，在处理STOMP消息的时候可能会用到它们。

表**18.1　Spring**能够使用某一个消息转换器将消息负载转换为**Java**类型

消息转换器	描　　述
ByteArrayMessageConverter	实现MIME类型为“application/octet-stream”的消息与byte[]之间的相互转换
MappingJackson2MessageConverter	实现MIME类型为“application/json”的消息与Java对象之间的相互转换
StringMessageConverter	实现MIME类型为“text/plain”的消息与String之间的相互转换

假设handleShout()方法所处理消息的内容类型为“application/json”（这应该是一个安全的假设，因为Shout不是byte[]和String），MappingJackson2MessageConverter会负责将JSON消息转换为Shout对象。就像在HTTP中对应的MappingJackson2HttpMessageConverter一样，MappingJackson2MessageConverter会将其任务委托给底层的Jackson 2 JSON处理器。默认情况下，Jackson会使用反射将JSON属性映射为Java对象的属性。尽管在本例中没有必要，但是我们可以通过在Java类型上使用Jackson注解，影响具体的转换行为。

处理订阅

除了@MessagingMapping注解以外，Spring还提供了@SubscribeMapping注解。与@MessagingMapping注解方法类似，当收到STOMP订阅消息的时候，带有@SubscribeMapping注解的方法将会触发。

很重要的一点，与@MessagingMapping方法类似，@SubscribeMapping方法也是通过

AnnotationMethodMessageHandler接收消息的（如图18.2和图18.3所示）。按照程序清单18.5的配置，这就意味着@SubscribeMapping方法只能处理目的地以“/app”为前缀的消息。

这可能看上去有些诡异，因为应用发出的消息都会经过代理，目的地要以“/topic”或“/queue”打头。客户端会订阅这些目的地，而不会订阅前缀为“/app”的目的地。如果客户端订阅“/topic”和“/queue”这样的目的地，那么@SubscribeMapping方法也就无法处理这样的订阅了。如果是这样的话，@SubscribeMapping有什么用处呢？

@SubscribeMapping的主要应用场景是实现请求-回应模式。在请求-回应模式中，客户端订阅某一个目的地，然后预期在这个目的地上获得一个一次性的响应。

例如，考虑如下@SubscribeMapping注解标注的方法：

```
@SubscribeMapping({"/marco"})
public Shout handleSubscription() {
  Shout outgoing = new Shout();
  outgoing.setMessage("Polo!");
  return outgoing;
}
```

可以看到，handleSubscription()方法使用了@SubscribeMapping注解，用这个方法来处理对“/app/marco”目的地的订阅（与@MessageMapping类似，“/app”是隐含的）。当处理这个订阅时，handleSubscription()方法会产生一个输出的Shout对象并将其返回。然后，Shout对象会转换成一条消息，并且会按照客户端订阅时相同的目的地发送回客户端。

如果你觉得这种请求-回应模式与HTTP GET的请求-响应模式并没有太大差别的话，那么你基本上是正确的。但是，这里的关键区别在于HTTP GET请求是同步的，而订阅的请求-回应模式则是异步的，这样客户端能够在回应可用时再去处理，而不必等待。

编写**JavaScript**客户端

handleShout()方法已经可以处理发送过来的消息了。现在，我们需要的就是发送消息的客户端。

如下的程序清单展现了一些JavaScript客户端代码，它会连接“/marcopolo”端点并发送“Marco!”消息。

程序清单**18.7**　借助**STOMP**库，通过**JavaScript**发送消息

```
var url = 'http://' + window.location.host + '/stomp/marcopolo';
var sock = new SockJS(url);                          <-- 创建 SockJS 连接

var stomp = Stomp.over(sock);                         <-- 创建 STOMP 客户端

var payload = JSON.stringify({ 'message': 'Marco!' });

stomp.connect('guest', 'guest', function(frame) {    <-- 连接 STOMP 端点
   stomp.send("/marco", {}, payload);          <-- 发送消息
 });
```

与我们之前的JavaScript客户端样例类似，在这里首先针对给定的URL创建一个`SockJS`实例。在本例中，URL引用的是程序清单18.5中所配置的STOMP端点（不包括应用的上下文路径"/stomp"）。

但是，这里的区别在于，我们不再直接使用SockJS，而是通过调用`Stomp.over(sock)`创建了一个STOMP客户端实例。这实际上封装了`SockJS`，这样就能在WebSocket连接上发送STOMP消息。

接下来，我们使用STOMP进行连接，假设连接成功，然后发送带有JSON负载的消息到名为"/marco"的目的地。往`send()`方法传递的第二个参数是一个头信息的Map，它会包含在STOMP的帧中，不过在这个例子中，我们没有提供任何参数，Map是空的。

现在，我们有了能够发送消息到服务器的客户端，以及用来处理消息的服务端处理器方法。这是一个好的开端，但是你可能已经发现这都是单向的。接下来，我们让服务器发出的声音，看一下如何发送消息给客户端。

18.3.3　发送消息到客户端

到目前为止，客户端负责了所有的消息发送，服务器只能监听这些消息。对于WebSocket和STOMP来说，这是一种合法的用法，但是当你考虑使用WebSocket的时候，所设想的使用场景恐怕并非如此。WebSocket通常视为服务器发送数据给浏览器的一种方式，采用这种方式所发送的数据不必位于HTTP请求的响应中。使用Spring和WebSocket/STOMP的话，该如何与基于浏览器的客户端通信呢？

Spring提供了两种发送数据给客户端的方法：

- 作为处理消息或处理订阅的附带结果；
- 使用消息模板。

我们已经了解了一些处理消息和处理订阅的方法，所以首先看一下如何通过这些方法发送消息给客户端。然后，再看一下Spring的SimpMessagingTemplate，它能够在应用的任何地方发送消息。

在处理消息之后，发送消息

程序清单18.6中，handleShout()只是简单地返回void。它的任务就是处理消息，并不需要给客户端回应。

如果你想要在接收消息的时候，同时在响应中发送一条消息，那么需要做的仅仅是将内容返回就可以了，方法签名不再是使用void。例如，如果你想发送“Polo!”消息作为“Marco!”消息的回应，那么只需将handleShout()修改为如下所示：

```
@MessageMapping("/marco")
public Shout handleShout(Shout incoming) {
  logger.info("Received message: " + incoming.getMessage());

  Shout outgoing = new Shout();
  outgoing.setMessage("Polo!");
  return outgoing;
}
```

在这个新版本的handleShout()方法中，会返回一个新的Shout对象。通过简单地返回一个对象，处理器方法同时也变成了发送方法。当@MessageMapping注解标示的方法有返回值的时候，返回的对象将会进行转换（通过消息转换器）并放到STOMP帧的负载中，然后发送给消息代理。

默认情况下，帧所发往的目的地会与触发处理器方法的目的地相同，只不过会添加上“/topic”前缀。就本例而言，这意味着handleShout()方法所返回的Shout对象会写入到STOMP帧的负载中，并发布到“/topic/marco”目的地。不过，我们可以通过为方法添加@SendTo注解，重载目的地：

```
@MessageMapping("/marco")
@SendTo("/topic/shout")
public Shout handleShout(Shout incoming) {
  logger.info("Received message: " + incoming.getMessage());

  Shout outgoing = new Shout();
  outgoing.setMessage("Polo!");
  return outgoing;
}
```

按照这个@SendTo注解，消息将会发布到“/topic/shout”。所有订阅这个主题的应用（如客户端）都会收到这条消息。

这样的话，handleShout()在收到一条消息的时候，作为响应也会发送一条消息。按照类似的方式，@SubscribeMapping注解标注的方式也能发送一条消息，作为订阅的回应。例如，通过为控制器添加如下的方法，当客户端订阅的时候，将会发送一条Shout信息：

```
@SubscribeMapping("/marco")
public Shout handleSubscription() {
  Shout outgoing = new Shout();
  outgoing.setMessage("Polo!");
  return outgoing;
}
```

这里的@SubscribeMapping注解表明当客户端订阅“/app/marco”（“/app”是应用目的地的前缀）目的地的时候，将会调用handleSubscription()方法。它所返回的Shout对象将会进行转换并发送回客户端。

@SubscribeMapping的区别在于这里的Shout消息将会直接发送给客户端，而不必经过消息代理。如果你为方法添加@SendTo注解的话，那么消息将会发送到指定的目的地，这样会经过代理。

在应用的任意地方发送消息

@MessageMapping和@SubscribeMapping提供了一种很简单的方式来发送消息，这是接收消息或处理订阅的附带结果。不过，Spring的SimpMessagingTemplate能够在应用的任何地方发送消息，甚至不必以首先接收一条消息作为前提。

使用SimpMessagingTemplate的最简单方式是将它（或者其接口SimpMessage-SendingOperations）自动装配到所需的对象中。

为了将这一切付诸实施，我们重新看一下Spittr的首页，为其提供实时的Spittle feed功能。按照其当前的写法，控制器会处理首页的请求，将最新的Spittle列表获取到，并将其放到模型中，然后渲染到用户的浏览器中。尽管这样运行起来也不错，但是它并没有提供Spittle更新的实时feed。如果用户想要看一个更新的Spittle feed，那必须要在浏览器中刷新页面。

我们不必要求用户刷新页面，而是让首页订阅一个STOMP主题，在Spittle创建的时候，该主题能够收到Spittle更新的实时feed。在首页中，我们需要添加如下的JavaScript代码块：

```
<script>
  var sock = new SockJS('spittr');
  var stomp = Stomp.over(sock);

  stomp.connect('guest', 'guest', function(frame) {
    console.log('Connected');
    stomp.subscribe("/topic/spittlefeed", handleSpittle);
  });

  function handleSpittle(incoming) {
    var spittle = JSON.parse(incoming.body);
    console.log('Received: ', spittle);
    var source = $("#spittle-template").html();
    var template = Handlebars.compile(source);
    var spittleHtml = template(spittle);
    $('.spittleList').prepend(spittleHtml);
  }
</script>
```

与之前的样例一样，我们首先创建了SockJS实例，然后基于该SockJS实例创建了Stomp实例。在连接到STOMP代理之后，我们订阅了“/topic/spittlefeed”，并指定当消息达到的时候，由handleSpittle()函数来处理Spittle更新。handleSpittle()函数会将传入的消息体解析为对应的JavaScript对象，然后使用Handlebars库将Spittle数据渲染为HTML并插入到列表中。Handlebars模板定义在一个单独的<script>标签中，如下所示：

```
<script id="spittle-template" type="text/x-handlebars-template">
  <li id="preexist">
  <div class="spittleMessage">{{message}}</div>
  <div>
   <span class="spittleTime">{{time}}</span>
   <span class="spittleLocation">({{latitude}}, {{longitude}})</span>
  </div>
  </li>
</script>
```

在服务器端，我们可以使用SimpMessagingTemplate将所有新创建的Spittle以消息的形式发布到“/topic/spittlefeed”主题上。如下程序清单展现的SpittleFeedServiceImpl就是实现该功能的简单服务：

程序清单**18.8　SimpMessagingTemplate**能够在应用的任何地方发布消息

```
package spittr;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.simp.SimpMessageSendingOperations;
import org.springframework.stereotype.Service;

@Service
public class SpittleFeedServiceImpl implements SpittleFeedService {

  private SimpMessageSendingOperations messaging;

  @Autowired
  public SpittleFeedServiceImpl(
        SimpMessageSendingOperations messaging) {    <— 注入消息模板
     this.messaging = messaging;
  }

  public void broadcastSpittle(Spittle spittle) {
    messaging.convertAndSend("/topic/spittlefeed", spittle); <— 发送消息
   }

}
```

配置Spring支持STOMP的一个副作用就是在Spring应用上下文中已经包含了SimpMessagingTemplate。因此，我们在这里没有必要再创建新的实例。Spittle-FeedServiceImpl的构造器使用了@Autowired注解，这样当创建SpittleFeedService-Impl的时候，就能注入SimpMessagingTemplate（以SimpMessageSendingOperations的形式）了。

发送Spittle消息的地方在broadcastSpittle()方法中。它在注入的SimpMessageSendingOperations上调用了convertAndSend()方法，将Spittle转换为消息，并将其发送到“/topic/spittlefeed”主题上。如果你觉得convertAndSend()方法看起来很眼熟的话，那是因为它模拟了JmsTemplate和RabbitTemplate所提供的同名方法。

不管我们通过convertAndSend()方法，还是借助处理器方法的结果，在发布消息给STOMP主题的时候，所有订阅该主题的客户端都会收到消息。在这个场景下，我们希望所有的客户端都能及时看到实时的Spittle feed，这种做法是很好的。但有的时候，我们希望发送消息给指定的用户，而不是所有的客户端。

18.4 为目标用户发送消息

到目前为止，我们所发送和接收的消息都是客户端（在Web浏览器中）和服务器端之间的，并没有考虑到客户端的用户。当带有@MessageMapping注解的方法触发时，我们知道收到了消息，但是并不知道消息来源于谁。类似地，因为我们不知道用户是谁，所以消息会发送到所有订阅对应主题的客户端上，没有办法发送消息给指

定用户。

但是，如果你知道用户是谁的话，那么就能处理与某个用户相关的消息，而不仅仅是与所有客户端相关联。好消息是我们已经了解了如何识别用户。通过使用与第9章相同的认证机制，我们可以使用Spring Security来认证用户，并为目标用户处理消息。

在使用Spring和STOMP消息功能的时候，我们有三种方式利用认证用户：

- @MessageMapping和@SubscribeMapping标注的方法能够使用Principal来获取认证用户；
- @MessageMapping、@SubscribeMapping和@MessageException方法返回的值能够以消息的形式发送给认证用户；
- SimpMessagingTemplate能够发送消息给特定用户。

我们首先看一下前两种方式，它们都能让控制器的消息处理方法使用针对特定用户的消息。

18.4.1 在控制器中处理用户的消息

如前所述，在控制器的@MessageMapping或@SubscribeMapping方法中，处理消息时有两种方式了解用户信息。在处理器方法中，通过简单地添加一个Principal参数，这个方法就能知道用户是谁并利用该信息关注此用户相关的数据。除此之外，处理器方法还可以使用@SendToUser注解，表明它的返回值要以消息的形式发送给某个认证用户的客户端（只发送给该客户端）。

为了阐述该功能，让我们编写一个控制器方法，它会根据传入的消息创建新的Spittle对象，并发送一个回应，表明Spittle已经保存成功。如果你觉得这个场景很熟悉的话，那是因为在第16章我们以REST端点的形式实现了它。但是REST请求是同步的，当服务器处理的时候，客户端必须要等待。通过将Spittle发送为STOMP消息，我们可以充分发挥STOMP消息异步的优势。

考虑如下的handleSpittle()方法，它会处理传入的消息并将其存储为Spittle：

```
@MessageMapping("/spittle")
@SendToUser("/queue/notifications")
public Notification handleSpittle(
      Principal principal, SpittleForm form) {

  Spittle spittle = new Spittle(
      principal.getName(), form.getText(), new Date());

  spittleRepo.save(spittle);

  return new Notification("Saved Spittle");
}
```

可以看到，handleSpittle()方法接受Principal对象和SpittleForm对象作为参数。它使用这两个对象创建一个Spittle实例并借助SpittleRepository将实例保存起来。最后，它返回一个新的Notification，表明Spittle已经保存成功。

当然，比起方法内部的功能，这个方法体外部所做事情也许更让我们感兴趣。因为这个方法使用了@MessageMapping注解，因此当有发往“/app/spittle”目的地的消息到达时，该方法就会触发，并且会根据消息创建SpittleForm对象，如果用户已经认证过的话，将会根据STOMP帧上的头信息得到Principal对象。

但是，需要特别关注的是，返回的Notification到哪里去了。@SendToUser注解指定返回的Notification要以消息的形式发送到“/queue/notifications”目的地上。在表面上，“/queue/notifications”并没有与特定用户关联。但因为这里使用的是@SendToUser注解而不是@SendTo，所以就会发生更多的事情了。

为了理解Spring如何发布消息，让我们先退后一步，看一下针对控制器方法发布Notification对象的目的地，客户端该如何进行订阅。考虑如下的这行JavaScript代码，它订阅了一个用户特定的目的地：

```
stomp.subscribe("/user/queue/notifications", handleNotifications);
```

注意，这个目的地使用了“/user”作为前缀，在内部，以“/user”作为前缀的目的地将会以特殊的方式进行处理。这种消息不会通过AnnotationMethodMessageHandler（像应用消息那样）来处理，也不会通过SimpleBrokerMessageHandler或StompBrokerRelayMessageHandler（像代理消息那样）来处理，以“/user”为前缀的消息将会通过UserDestinationMessageHandler进行处理，如图18.4所示。

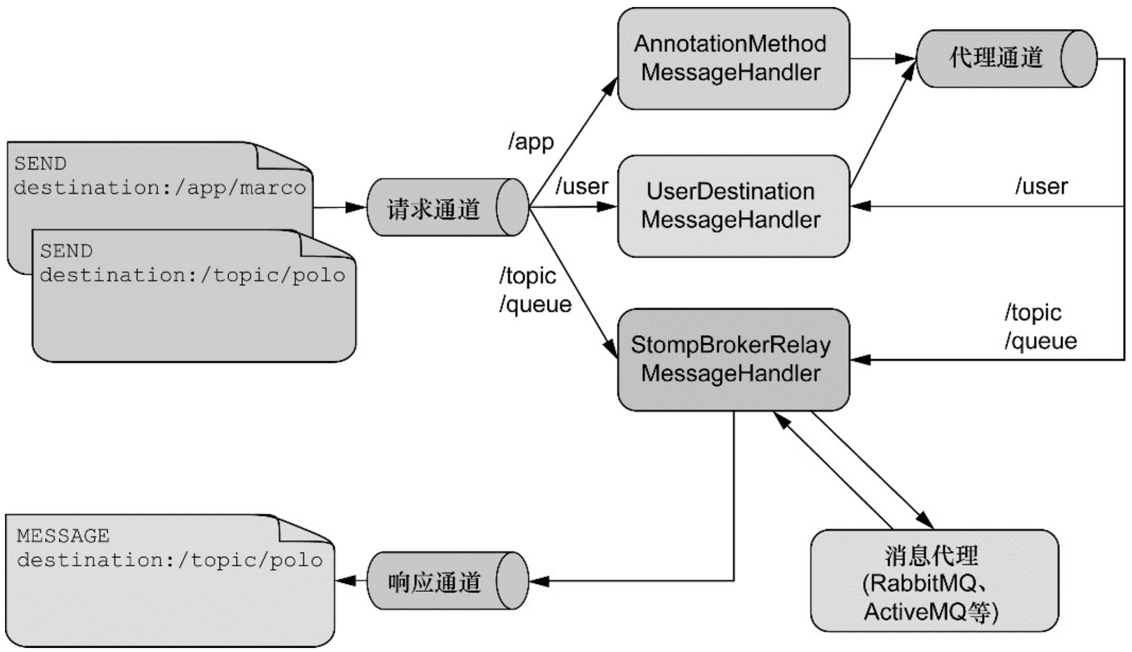

图18.4　用户消息流会通过UserDestinationMessageHandler进行处理，它会将消息重路由到某个用户独有的目的地上

`UserDestinationMessageHandler`的主要任务是将用户消息重新路由到某个用户独有的目的地上。在处理订阅的时候，它会将目标地址中的“/user”前缀去掉，并基于用户的会话添加一个后缀。例如，对“/user/queue/notifications”的订阅最后可能路由到名为“/queue/notifications-user6hr83v6t”的目的地上。

在我们的样例中，`handleSpittle()`方法使用了`@SendToUser("/queue/notifications")`注解。这个新的目的地以“/queue”作为前缀，根据配置，这是`StompBrokerRelayMessageHandler`（或`SimpleBrokerMes`要处理的前缀，所以消息接下来会到达这里。最终，客户端会订阅这个目的地，因此客户端会收到Notification消息。

在控制器方法中，`@SendToUser`注解和`Principal`参数是很有用的。但是在程序清单18.8中，我们看到借助消息模板，可以在应用的任何位置发送消息。接下来看一下如何使用`SimpMessagingTemplate`将消息发送给特定用户。

18.4.2　为指定用户发送消息

除了`convertAndSend()`以外，`SimpMessagingTemplate`还提供了`convertAndSendToUser()`方法。按照名字就可以判断出来，`convertAndSendToUser()`方法能够让我们给特定用户发送

消息。

为了阐述该功能，我们要在Spittr应用中添加一项特性，当其他用户提交的Spittle提到某个用户时，将会提醒该用户。例如，如果Spittle文本中包含“@jbauer”，那么我们就应该发送一条消息给使用“jbauer”用户名登录的客户端。如下程序清单中的broadcast-Spittle()方法使用了convertAndSendToUser()，从而能够提醒所谈论到的用户。

程序清单**18.9　convertAndSendToUser()**能够发送消息给特定用户

```
package spittr;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.stereotype.Service;

@Service
public class SpittleFeedServiceImpl implements SpittleFeedService {

  private SimpMessagingTemplate messaging;
  private Pattern pattern = Pattern.compile("\\@(\\S+)");   <-- 实现用户提及功能的正则表达式

  @Autowired
  public SpittleFeedServiceImpl(SimpMessagingTemplate messaging) {
    this.messaging = messaging;
  }

  public void broadcastSpittle(Spittle spittle) {

    messaging.convertAndSend("/topic/spittlefeed", spittle);

    Matcher matcher = pattern.matcher(spittle.getMessage());
    if (matcher.find()) {
      String username = matcher.group(1);
      messaging.convertAndSendToUser(                <-- 发送提醒给用户
           username, "/queue/notifications",
          new Notification("You just got mentioned!"));
    }
  }

}
```

在broadcastSpittle()中，如果给定Spittle对象的消息中包含了类似于用户名的内容（也就是以“@”开头的文本），那么一个新的Notification将会发送到名为“/queue/notifications”的目的地上。因此，如果Spittle中包含“@jbauer”的话，Notification将会发送到“/user/jbauer/queue/notifications”目的地上。

18.5　处理消息异常

有时候，事情并不会按照我们预期的那样发展。在处理消息的时候，有可能会出错并抛出异常。因为STOMP消息异步的特点，发送者可

能永远也不会知道出现了错误。除了Spring的日志记录以外，异常有可能会丢失，没有资源或机会恢复。

在Spring MVC中，如果在请求处理中，出现异常的话，@ExceptionHandler方法将有机会处理异常。与之类似，我们也可以在某个控制器方法上添加@MessageException-Handler注解，让它来处理@MessageMapping方法所抛出的异常。

例如，考虑如下的方法，它会处理消息方法所抛出的异常：

```
@MessageExceptionHandler
public void handleExceptions(Throwable t) {
  logger.error("Error handling message: " + t.getMessage());
}
```

按照最简单的形式，@MessageExceptionHandler标注的方法能够处理消息方法中所抛出的异常。但是，我们也可以以参数的形式声明它所能处理的异常：

```
@MessageExceptionHandler(SpittleException.class)
public void handleExceptions(Throwable t) {
  logger.error("Error handling message: " + t.getMessage());
}
```

或者，以数组参数的形式指定多个异常类型：

```
@MessageExceptionHandler(
        {SpittleException.class, DatabaseException.class})
public void handleExceptions(Throwable t) {
  logger.error("Error handling message: " + t.getMessage());
}
```

尽管它只是以日志的方式记录了所发生的错误，但是这个方法可以做更多的事情。例如，它可以回应一个错误：

```
@MessageExceptionHandler(SpittleException.class)
@SendToUser("/queue/errors")
public SpittleException handleExceptions(SpittleException e) {
  logger.error("Error handling message: " + e.getMessage());
  return e;
}
```

在这里，如果抛出SpittleException的话，将会记录这个异常，然后将其返回。在18.4.1小节中，我们已经学过，UserDestinationMessageHandler会重新路由这个消息到特定用户所对应的唯一路径。

18.6　小结

如果在应用间发送消息的话，那WebSocket是一种令人兴奋的通信方式，尤其是如果其中某个应用运行在Web浏览器中更是如此。当编写存在大量交互的Web应用程序时，它是很重要的，能够实现从服务器无缝的发送和接收数据。

Spring对WebSocket的支持包括低层级的API，它能够让我们使用原始的WebSocket连接。但是，WebSocket并没有在Web浏览器、服务器以及网络代理上得到广泛支持。因此，Spring同时还支持SockJS，这个协议能够在WebSocket不可用的时候提供备用的通信模式。

Spring还提供了高级的编程模型，也就是使用STOMP线路级协议来处理WebSocket消息。在这个更高级的模型中，能够在Spring MVC控制器中处理STOMP消息，类似于处理HTTP消息的方式。

在过去的两章中，我们看到了多种在应用间异步发送消息的方式。Spring还有另外一种处理异步消息的方式。在下一章中，我们将会看到如何使用Spring发送Email。

第19章　使用Spring发送Email

本章内容：

- 配置Spring的Email抽象功能
- 发送丰富内容的Email消息
- 使用模板构建Email消息

毫无疑问，Email已经成为常见的通信形式，取代了很多传统的通信方式，如邮政邮件、电话，在一定程度上也替代了面对面的交流。Email能够提供了与第17章中所讨论的异步消息相同的收益，只不过发送者和接收者都是实际的人而已。只要你在邮件客户端上点击“发送”按钮，就可以转移到其他的任务中了，因为我们知道接收者最终将会收到并阅读（希望如此）你的Email。

但是，Email的发送者不一定是实际的人。有时候，Email消息是由应用程序发送给用户的。有可能是电子商务网站上的订单确认邮件，也有可能是银行账户某项交易的自动提醒。不管邮件的主题是什么，我们都可能需要开发发送Email消息的应用程序。幸好，在这个方面，Spring会为我们提供帮助。

在第17章中，我们借助Spring对消息功能的支持，以排队任务的形式异步发送Spittle提醒给Spittr的其他用户。但是，这项任务并未完成，因为没有发送Email消息。现在，我们将会完成这项任务，在本章首先会看一下Spring是如何抽象邮件发送这一问题的，然后利用这一抽象发送包含Spittle提醒的Email消息。

19.1　配置Spring发送邮件

Spring Email抽象的核心是`MailSender`接口。顾名思义，`MailSender`的实现能够通过连接Email服务器实现邮件发送的功能，如图19.1所示。

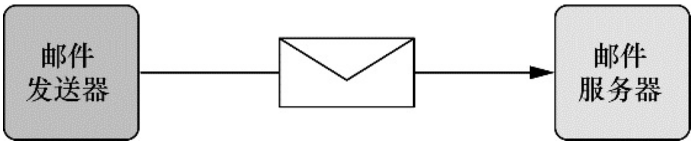

图19.1　Spring的MailSender接口是Spring Email抽象API的核心组件。

它把Email发送给邮件服务器，由服务器进行邮件投递

Spring自带了一个MailSender的实现也就是JavaMailSenderImpl，它会使用JavaMail API来发送Email。Spring应用在发送Email之前，我们必须要将JavaMailSenderImpl装配为Spring应用上下文中的一个bean。

19.1.1 配置邮件发送器

按照最简单的形式，我们只需在@Bean方法中使用几行代码就能将JavaMailSenderImpl配置为一个bean：

```
@Bean
public MailSender mailSender(Environment env) {
  JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
  mailSender.setHost(env.getProperty("mailserver.host"));
  return mailSender;
}
```

属性host是可选的（它默认是底层JavaMail会话的主机），但你可能希望设置该属性。它指定了要用来发送Email的邮件服务器主机名。按照这里的配置，会从注入的Environment中获取值，这样我们就能在Spring之外管理邮件服务器的配置（比如在属性文件中）。

默认情况下，JavaMailSenderImpl假设邮件服务器监听25端口（标准的SMTP端口）。如果你的邮件服务器监听不同的端口，那么可以使用port属性指定正确的端口号。例如：

```
@Bean
public MailSender mailSender(Environment env) {
  JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
  mailSender.setHost(env.getProperty("mailserver.host"));
  mailSender.setPort(env.getProperty("mailserver.port"));
  return mailSender;
}
```

类似地，如果邮件服务器需要认证的话，你还需要设置username和password属性：

```
@Bean
public MailSender mailSender(Environment env) {
  JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
  mailSender.setHost(env.getProperty("mailserver.host"));
  mailSender.setPort(env.getProperty("mailserver.port"));
  mailSender.setUsername(env.getProperty("mailserver.username"));
  mailSender.setPassword(env.getProperty("mailserver.password"));
  return mailSender;
}
```

到目前为止，JavaMailSenderImpl已经配置完成，它可以创建自己的邮件会话，但是你可能已经在JNDI中配置了javax.mail.MailSession（也可能是你的应用服务器放在那里的）。如果这样的话，那就没有必要为JavaMailSenderImpl配置详细的服务器细节了。我们可以配置它使用JNDI中已就绪的MailSession。

借助JndiObjectFactoryBean，我们可以在如下的@Bean方法中配置一个bean，它会从JNDI中查找MailSession：

```
@Bean
public JndiObjectFactoryBean mailSession() {
  JndiObjectFactoryBean jndi = new JndiObjectFactoryBean();
  jndi.setJndiName("mail/Session");
  jndi.setProxyInterface(MailSession.class);
  jndi.setResourceRef(true);
  return jndi;
}
```

我们已经看到过如何使用Spring的<jee:jndi-lookup>元素从JNDI中获取对象，这里可以使用<jee:jndi-lookup>来创建一个bean，它引用了JNDI中的邮件会话：

```
<jee:jndi-lookup id="mailSession"
  jndi-name="mail/Session" resource-ref="true" />
```

邮件会话准备就绪之后，我们现在可以将其装配到mailSender bean中了：

```
@Bean
public MailSender mailSender(MailSession mailSession) {
  JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
  mailSender.setSession(mailSession);
  return mailSender;
}
```

通过将邮件会话装配到JavaMailSenderImpl的session属性中，我们已经完全替换了原来的服务器（以及用户名/密码）配置。现在邮件会话完全通过JNDI进行配置和管理。JavaMailSenderImpl能够专注于发送邮件而不必自己处理邮件服务器了。

19.1.2 装配和使用邮件发送器

邮件发送器已经配置完成，现在需要将其装配到使用它的bean中了。在Spittr应用程序中，最适合发送Email的是SpitterEmailServiceImpl类。这个类有一个mailSender属

性，它使用了`@Autowired`注解：

```
@Autowired
JavaMailSender mailSender;
```

当Spring将`SpitterEmailServiceImpl`创建为一个bean的时候，它将查找实现了`MailSender`的bean，这样的bean可以装配到`mailSender`属性中。它将会找到我们在前边配置的`mailSender`bean并使用它。`mailSender`bean装配完成后，我们就可以构建和发送Email了。

我们想要给Spitter用户发送Email提示他的朋友写了新的Spittle，所以我们需要一个方法来发送Email，这个方法要接受Email地址和`Spittle`对象信息。如下的`sendSimpleSpittleEmail()`方法使用邮件发送器完成了该功能：

程序清单**19.1**　使用**Spring**的**MailSender**发送**Email**

```
public void sendSimpleSpittleEmail(String to, Spittle spittle) {
  SimpleMailMessage message = new SimpleMailMessage();          构造消息
  String spitterName = spittle.getSpitter().getFullName();
  message.setFrom("noreply@spitter.com");                        Email 地址
  message.setTo(to);
  message.setSubject("New spittle from " + spitterName);
  message.setText(spitterName + " says: " +                      设置消息文本
          spittle.getText());

  mailSender.send(message);                                      发送 Email
```

`sendSimpleSpittleEmail()`方法所做的第一件事就是构造`SimpleMailMessage`实例。正如其名称所示，这个对象可以很便捷地发送Email消息。

接下来，将设置消息的细节。通过邮件消息的`setFrom()`和`setTo()`方法指定了Email的发送者和接收者。在通过`setSubject()`方法设置完主题后，虚拟的“信封”已经完成了。剩下的就是调用`setText()`方法来设置消息的内容。

最后一步是将消息传递给邮件发送器的`send()`方法，这样邮件就发送出去了。

现在，我们已经配置好了邮件发送器并使用它来发送简单的Email消息。可以看到，使用Spring的Email抽象非常简单。我们可以到此为止并转到下一章，但是如果这样的话将会错过Spring Email抽象中很有意思的内容。让我们更进一步，看一下如何添加附件并创建丰富内容的Email消息。

19.2 构建丰富内容的Email消息

对于简单的事情来讲，纯文本的Email消息是比较合适的，比如邀请朋友去观看比赛。但是，如果你要发送照片或文档的话，这种方式就不那么理想了。如果作为市场推广Email的话，它也无法吸引接收者的注意。

幸好，Spring的Email功能并不局限于纯文本的Email。我们可以添加附件，甚至可以使用HTML来美化消息体的内容。让我们首先从基本的添加附件开始，然后更进一步，借助HTML使我们的Email消息更加美观。

19.2.1 添加附件

如果发送带有附件的Email，关键技巧是创建multipart类型的消息——Email由多个部分组成，其中一部分是Email体，其他部分是附件。

对于发送附件这样的需求来说，`SimpleMailMessage`过于简单了。为了发送multipart类型的Email，你需要创建一个MIME（Multipurpose Internet Mail Extensions）的消息，我们可以从邮件发送器的`createMimeMessage()`方法开始：

```
MimeMessage message = mailSender.createMimeMessage();
```

就这样，我们已经有了要使用的MIME消息。看起来，我们所需要做的就是指定收件人和发件人地址、主题、一些内容以及一个附件。尽管确实是这样，但并不是你想的那么简单。`javax.mail.internet.MimeMessage`本身的API有些笨重。好消息是，Spring提供的`MimeMessageHelper`可以帮助我们。

为了使用`MimeMessageHelper`，我们需要实例化它并将`MimeMessage`传给其构造器：

```
MimeMessageHelper helper = new MimeMessageHelper(message, true);
```

构造方法的第二个参数，在这里是个布尔值`true`，表明这个消息是multipart类型的。

得到了`MimeMessageHelper`实例后，我们就可以组装Email消息了。这里最主要区别在于使用helper的方法来指定Email细节，而不再是设置消息对象：

```
String spitterName = spittle.getSpitter().getFullName();
helper.setFrom("noreply@spitter.com");
helper.setTo(to);
helper.setSubject("New spittle from " + spitterName);
helper.setText(spitterName + " says: " + spittle.getText());
```

在发送Email之前，你唯一还要做的就是添加附件：在本例中，也就是一张图标图片。为了做到这一点，你需要加载图片并将其作为资源，然后将这个资源传递给helper的addAttachment方法：

```
FileSystemResource couponImage =
    new FileSystemResource("/collateral/coupon.png");
helper.addAttachment("Coupon.png", couponImage);
```

在这里，我们使用Spring的FileSystemResource来加载位于应用类路径下的coupon.png。然后，调用addAttachment()。第一个参数是要添加到Email中附件的名称，第二个参数是图片资源。

multipart类型的Email已经构建完成了，现在可以发送它了。完整的sendSpittleEmailWithAttachment()方法如下所示。

程序清单**19.2**　使用**MimeMessageHelper**发送带有附件的**Email**

```
public void sendSpittleEmailWithAttachment(
        String to, Spittle spittle) throws MessagingException {
  MimeMessage message = mailSender.createMimeMessage();
  MimeMessageHelper helper =
      new MimeMessageHelper(message, true);        <-- 构造消息 helper
   String spitterName = spittle.getSpitter().getFullName();
  helper.setFrom("noreply@spitter.com");
  helper.setTo(to);
  helper.setSubject("New spittle from " + spitterName);
  helper.setText(spitterName + " says: " + spittle.getText());
  FileSystemResource couponImage =
      new FileSystemResource("/collateral/coupon.png");
  helper.addAttachment("Coupon.png", couponImage);    <-- 添加附件
   mailSender.send(message);
}
```

multipart类型的Email能够实现很多的功能，添加附件只是其中之一。除此之外，通过将Email体指明为HTML，我们可以生成比简单文本更漂亮的Email。接下来，看一下如何使用MimeMessageHelper来发送更吸引人的Email。

19.2.2　发送富文本内容的Email

发送富文本的Email与发送简单文本的Email并没有太大区别。关键是将消息的文本设置为HTML。要做到这一点只需将HTML字符串传递给helper的setText()方法，并将第二个参数设置为true：

```
helper.setText("<html><body><img src='cid:spitterLogo'>" +
  "<h4>" + spittle.getSpitter().getFullName() + " says...</h4>" +
  "<i>" + spittle.getText() + "</i>" +
      "</body></html>", true);
```

第二个参数表明传递进来的第一个参数是HTML，所以需要对消息的内容类型进行相应的设置。

要注意的是，传递进来的HTML包含了一个<img>标签，用来在Email中展现Spittr应用程序的logo。src属性可以设置为标准的“http:”URL，以便于从Web中获取Spittr的logo。但在这里，我们将logo图片嵌入在了Email之中。值“cid:spitterLogo”表明在消息中会有一部分是图片并以spitterLogo来进行标识。

为消息添加嵌入式的图片与添加附件很类似。不过这次不再使用helper的addAttachment()方法，而是要调用addInline()方法：

```
ClassPathResource image =
               new ClassPathResource("spitter_logo_50.png");
helper.addInline("spitterLogo", image);
```

addInline的第一个参数表明内联图片的标识符——与<img>标签的src属性所指定的相同。第二个参数是图片的资源引用，这里使用ClassPathResource从应用程序的类路径中获取图片。

除了setText()方法稍微不同以及使用了addInline()方法以外，发送含有富文本内容的Email与发送带有附件的普通文本消息很类似。为了进行对比，以下是新的sendRichSpitterEmail()方法。

```
public void sendRichSpitterEmail(String to, Spittle spittle)
                               throws MessagingException {
MimeMessage message = mailSender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setFrom("noreply@spitter.com");
helper.setTo("craig@habuma.com");
helper.setSubject("New spittle from " +
        spittle.getSpitter().getFullName());
helper.setText("<html><body><img src='cid:spitterLogo'>" +     <— 设置HTML内容体
     "<h4>" + spittle.getSpitter().getFullName() + " says...</h4>" +
    "<i>" + spittle.getText() + "</i>" +
        "</body></html>", true);
ClassPathResource image =
     new ClassPathResource("spitter_logo_50.png");
helper.addInline("spitterLogo", image);          <— 添加内联图片
 mailSender.send(message);
}
```

现在你发送的Email带有富文本内容和嵌入式图片了！你可以到此为止并完全结束你的Email代码。但创建Email体时，使用字符串拼接的

办法来构建HTML消息依旧让我觉得美中不足。在结束Email话题之前，让我们看看如何用模板来代替字符串拼接消息。

19.3 使用模板生成Email

使用字符串拼接来构建Email消息的问题在于Email最终会是什么样子并不清晰。在你的大脑中解析HTML标签并想象它在渲染时会是什么样子是挺困难的。而将HTML混合在Java代码中又会使得这个问题更加复杂。如果能够将Email的布局抽取到一个模板中，而这个模板可以由美术设计师（可能是很讨厌Java代码的人）来完成将会是很棒的一件事。

我们需要与最终HTML接近的方式来表达Email布局，然后将模板转换成String并传递给helper的`setText()`方法。在将模板转换为String时，我们有多种模板方案可供选择，包括Apache Velocity和Thymeleaf。让我们看一下如何使用这两种方案创建富文本的Email消息，先从Velocity开始吧。

19.3.1 使用Velocity构建Email消息

Apache Velocity是由Apache提供的通用模板引擎。Velocity有挺长的历史了，并且已经应用于各种任务中，包括代码生成以及代替JSP。它还能用于格式化富文本Email消息，也就是我们在这里的用法。

为了使用Velocity对Email进行布局，我们需要将VelocityEngine装配到`SpitterEmailServiceImpl`中。Spring提供了一个名为`VelocityEngineFactoryBean`的工厂bean，它能够在Spring应用上下文中很便利地生成VelocityEngine。`VelocityEngineFactoryBean`的声明如下：

```
@Bean
public VelocityEngineFactoryBean velocityEngine() {
  VelocityEngineFactoryBean velocityEngine =
          new VelocityEngineFactoryBean();

  Properties props = new Properties();
  props.setProperty("resource.loader", "class");
  props.setProperty("class.resource.loader.class",
          ClasspathResourceLoader.class.getName());
  velocityEngine.setVelocityProperties(props);
  return velocityEngine;
}
```

`VelocityEngineFactoryBean`唯一要设置的属性是`velocityProperties`。在本例中，我们将其配置为从类路径下

加载Velocity模板（关于配置Velocity的更多细节，请查阅Velocity文档）。

现在，我们可以将Velocity引擎装配到SpitterEmailServiceImpl中。因为SpitterEmailServiceImpl是使用组件扫描实现自动注册的，我们可以使用@Autowired来自动装配velocityEngine属性：

```
@Autowired
VelocityEngine velocityEngine;
```

现在，velocityEngine属性可用了，我们可以使用它将Velocity模板转换为String，并作为Email文本进行发送。为了帮助我们完成这一点，Spring自带了VelocityEngineUtils来简化将Velocity模板与模型数据合并成String的工作。以下是我们可能的使用方式：

```
Map<String, String> model = new HashMap<String, String>();
model.put("spitterName", spitterName);
model.put("spittleText", spittle.getText());
String emailText = VelocityEngineUtils.mergeTemplateIntoString(
          velocityEngine, "emailTemplate.vm", model );
```

为了给处理模板做准备，我们首先创建了一个Map用来保存模板使用的模型数据。在前面字符串拼接的代码中，我们需要Spitter的全名及其Spittle的文本，这里也是一样。为了产生合并后的Email文本，我们只需调用VelocityEngineUtils的mergeTemplateIntoString()方法并将Velocity引擎、模板路径（相对于类路径根）以及模型Map传递进去。

在Java代码中剩下的事情就是得到合并后的Email文本，并将其传递给helper的setText()方法：

```
helper.setText(emailText, true);
```

模板位于类路径的根目录下，是一个名为emailTemplate.vm的文件，它看起来可能是这样的：

```
<html>
<body>
  <img src='cid:spitterLogo'>
  <h4>${spitterName} says...</h4>
  <i>${spittleText}</i>
</body>
</html>
```

你可以看到，模板文件比前面的字符串拼接版本读起来容易多了。因

此，它也更容易维护和编辑。图19.2给出了这种类型Email的一个示例。

在看到图19.2的效果后，我觉得有很多地方可以对模板进行优化从而使得Email看起来更漂亮。但是，我将它作为给读者的练习。

Velocity作为模板引擎已经存在好多年了，并且适用于很多种任务。但是，如第6章所示，一种新的模板方案正在变得日益流行。接下来，我们看一下如何使用Thymeleaf来构建Spittle Email消息。

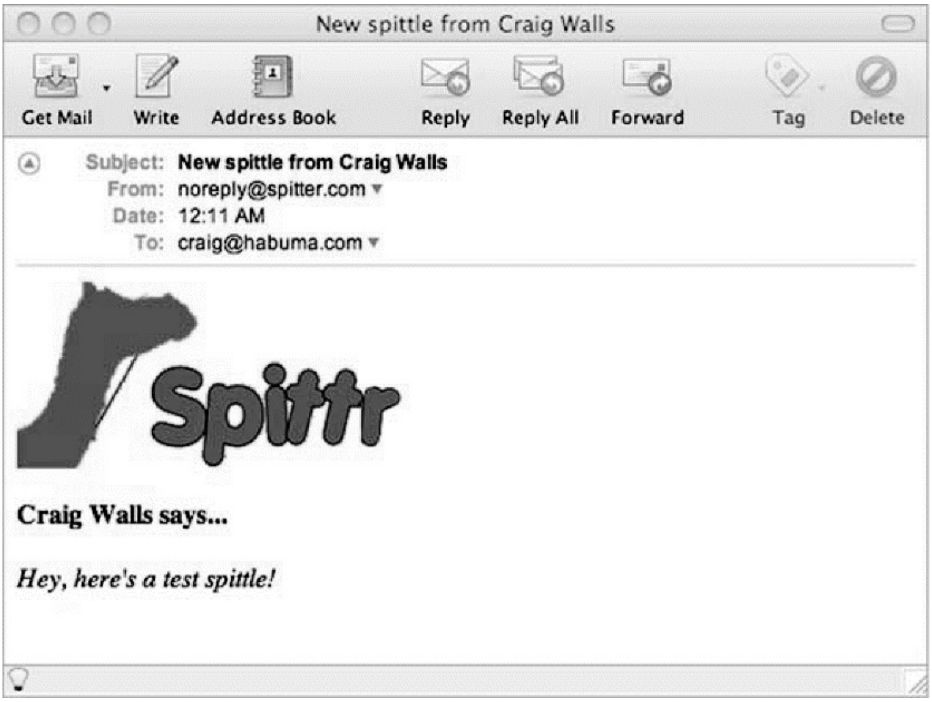

图19.2　Velocity模板和嵌入的图片能够装扮原本单调乏味的Email

19.3.2　使用Thymeleaf构建Email消息

如我们第6章所讨论的那样，Thymeleaf是一种很有吸引力的HTML模板引擎，因为它能够创建WYSIWYG的模板。与JSP和Velocity不同，Thymeleaf模板不包含任何特殊的标签库和特有的标签。这样模板设计师在工作的时候，能够使用任意他们所喜欢的HTML工具，而不必担心某个工具无法处理特定的标签。

当我们将Email模板转换为Thymeleaf模板时，Thymeleaf的WYSIWYG特性体现得非常明显：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
  <img src="spitterLogo.png" th:src='cid:spitterLogo'>
  <h4><span th:text="${spitterName}">Craig Walls</span> says...</h4>
  <i><span th:text="${spittleText}">Hello there!</span></i>
</body>
</html>
```

注意，这里没有任何自定义的标签（在JSP中可能会见到这种情况）。尽管模型属性是通过“`${}`”标记的，但是它们仅用于属性的值中，不会像Velocity那样用在外边。这种模板可以很容易地在Web浏览器中打开，并且以完整的形式进行展现，不必依赖于Thymeleaf引擎的处理。

使用Thymeleaf来生成和发送Email消息的做法非常类似于Velocity：

```
Context ctx = new Context();
ctx.setVariable("spitterName", spitterName);
ctx.setVariable("spittleText", spittle.getText());
String emailText = thymeleaf.process("emailTemplate.html", ctx);
...
helper.setText(emailText, true);
mailSender.send(message);
```

这里做的第一件事情就是创建Thymeleaf `Context`实例，并将模型数据填充进去。这与我们使用Velocity的时候，将模型数据填充到`Map`中很类似。然后，我们要求Thymeleaf处理模板，通过调用Thymeleaf引擎的`process()`方法，将上下文中的模型数据合并到模板中。最后，我们将结果形成的文本借助消息helper设置到Email消息中，并使用邮件发送器将消息发送出去。

这看起来很简单。但是Thymeleaf引擎（也就是`thymeleaf`变量）是从哪里来的呢？

这里的Thymeleaf引擎与我们在第6章构建Web视图时所使用的`SpringTemplateEngine`bean是相同的。在这里，我们使用构造器注入的方式将其注入到`SpitterEmailServiceImpl`中：

```
@Autowired
private SpringTemplateEngine thymeleaf;

@Autowired
public SpitterEmailServiceImpl(SpringTemplateEngine thymeleaf) {
  this.thymeleaf = thymeleaf;
}
```

不过，我们必要要对`SpringTemplateEngine`bean做一点小修改。在第6章中，它配置为从Servlet上下文中解析模板，而我们的Email模

板需要从类路径中解析。所以，除了ServletContextTemplateResolver，还需要一个ClassLoaderTemplateResolver：

```
@Bean
public ClassLoaderTemplateResolver emailTemplateResolver() {
  ClassLoaderTemplateResolver resolver =
          new ClassLoaderTemplateResolver();
  resolver.setPrefix("mail/");
  resolver.setTemplateMode("HTML5");
  resolver.setCharacterEncoding("UTF-8");
  setOrder(1);
  return resolver;
}
```

就大部分而言，配置ClassLoaderTemplateResolver bean的方式类似于ServletContextTemplateResolver。不过，需要注意，我们将prefix属性设置为“mail/”，这表明它会在类路径根的“mail”目录下开始查找Thymeleaf模板。因此，Email模板文件的名字必须是emailTemplate.html，并且位于类路径根的“mail”目录下。

因为我们现在有两个模板解析器，所以需要使用order属性表明优先使用哪一个。ClassLoaderTemplateResolver的order属性为1，因此我们修改一下ServletContext-TemplateResolver，将其order属性设置为2：

```
@Bean
public ServletContextTemplateResolver webTemplateResolver() {
  ServletContextTemplateResolver resolver =
          new ServletContextTemplateResolver();
  resolver.setPrefix("/WEB-INF/templates/");
  resolver.setTemplateMode("HTML5");
  resolver.setCharacterEncoding("UTF-8");
  setOrder(2);
  return resolver;
}
```

现在，剩下的任务就是修改SpringTemplateEnginebean的配置，让它使用这两个模板解析器：

```
@Bean
public SpringTemplateEngine templateEngine(
    Set<ITemplateResolver> resolvers) {
  SpringTemplateEngine engine = new SpringTemplateEngine();
  engine.setTemplateResolvers(resolvers);
  return engine;
}
```

在此之前，我们只有一个模板解析器，所以可以将其注入到SpringTemplateEngine的templateResolver属性中。但现在我们有了两个模板解析器，所以必须将它们作为Set的成员，然后

将这个`Set`注入到`templateResolvers`（复数）属性中。

19.4 小结

Email是人与人之间通信的重要形式，通常也是应用与人进行通信的一种形式。Spring基于Java所提供的Email功能，抽象了JavaMail，使得在Spring中使用和配置起来都更加简单。

在本章中，我们看到了如何使用Spring的Email抽象功能发送简单的Email消息，然后更进一步，学习了如何发送包含附件和经过HTML格式化的富文本消息。我们还看到了如何使用像Velocity和Thymeleaf这样的模板引擎生成富文本Email文本，避免了通过字符串拼接创建HTML。

在下一章中，我们将会学习如何借助Java管理扩展（Java Management Extensions，JMX）为Spring bean添加管理和通知功能。

第20章　使用JMX管理Spring Bean

本章内容：

- 将Spring bean暴露为MBean
- 远程管理Spring Bean
- 处理JMX通知

Spring对DI的支持是通过在应用中配置bean属性，这是一种非常不错的方法。不过，一旦应用已经部署并且正在运行，单独使用DI并不能帮助我们改变应用的配置。假设我们希望深入了解正在运行的应用并要在运行时改变应用的配置，此时，就可以使用Java管理扩展（Java Manage- ment Extensions，JMX）了。

JMX这项技术能够让我们管理、监视和配置应用。这项技术最初作为Java的独立扩展，从Java 5开始，JMX已经成为标准的组件。

使用JMX管理应用的核心组件是托管bean（managed bean，MBean）。所谓的MBean就是暴露特定方法的JavaBean，这些方法定义了管理接口。JMX规范定义了如下4种类型的MBean：

- 标准MBean：标准MBean的管理接口是通过在固定的接口上执行反射确定的，bean类会实现这个接口；
- 动态MBean：动态MBean的管理接口是在运行时通过调用`DynamicMBean`接口的方法来确定的。因为管理接口不是通过静态接口定义的，因此可以在运行时改变；
- 开放MBean：开放MBean是一种特殊的动态MBean，其属性和方法只限定于原始类型、原始类型的包装类以及可以分解为原始类型或原始类型包装类的任意类型；
- 模型MBean：模型MBean也是一种特殊的动态MBean，用于充当管理接口与受管资源的中介。模型Bean并不像它们所声明的那样来编写。它们通常通过工厂生成，工厂会使用元信息来组装管理接口。

Spring的JMX模块可以让我们将Spring bean导出为模型MBean，这样我们就可以查看应用程序的内部情况并且能够更改配置——甚至在应用

的运行期。接下来，将会介绍如何使用Spring对JMX的支持来管理Spring应用上下文中的bean。

20.1 将Spring bean导出为MBean

这里有几种方式可以让我们通过使用JMX来管理Spittr应用中的bean。为了让事情尽量保持简单，我们对程序清单5.10中SpittleController只做适度的改变，增加一个新的spittlesPerPage属性：

```
public static final int DEFAULT_SPITTLES_PER_PAGE = 25;
private int spittlesPerPage = DEFAULT_SPITTLES_PER_PAGE;

public void setSpittlesPerPage(int spittlesPerPage) {
  this.spittlesPerPage = spittlesPerPage;
}

public int getSpittlesPerPage() {
  return spittlesPerPage;
}
```

之前，当我们调用SpitterService的getRecentSpittles()方法时，SpittleController传入20作为第二个参数，这会查询最近的20条Spittle。现在，不再是在构建应用时通过硬编码进行决策，而是通过使用JMX在运行时进行决策。新增的spittlesPerPage属性只是第一步而已。

但是spittlesPerPage属性本身并不能实现通过外部配置来改变页面上所显示Spittle的数量。它只是bean的一个属性，跟bean的其他属性一样。我们下一步需要做的是把SpittleControllerbean暴露为MBean，而spittlePerPage属性将成为MBean的托管属性（managed attribute）。这时，我们就可以在运行时改变该属性的值。

Spring的MBeanExporter是将Spring Bean转变为MBean的关键。MBeanExporter可以把一个或多个Spring bean导出为MBean服务器（MBean server）内的模型 MBean。MBean服务器（有时候也被称为MBean代理）是MBean生存的容器。对MBean的访问，也是通过MBean服务器来实现的。

如图20.1所示，将Spring bean导出为JMX MBean之后，可以使用基于JMX的管理工具（例如JConsole或者VisualVM）查看正在运行的应用程序，显示bean的属性并调用bean的方法。

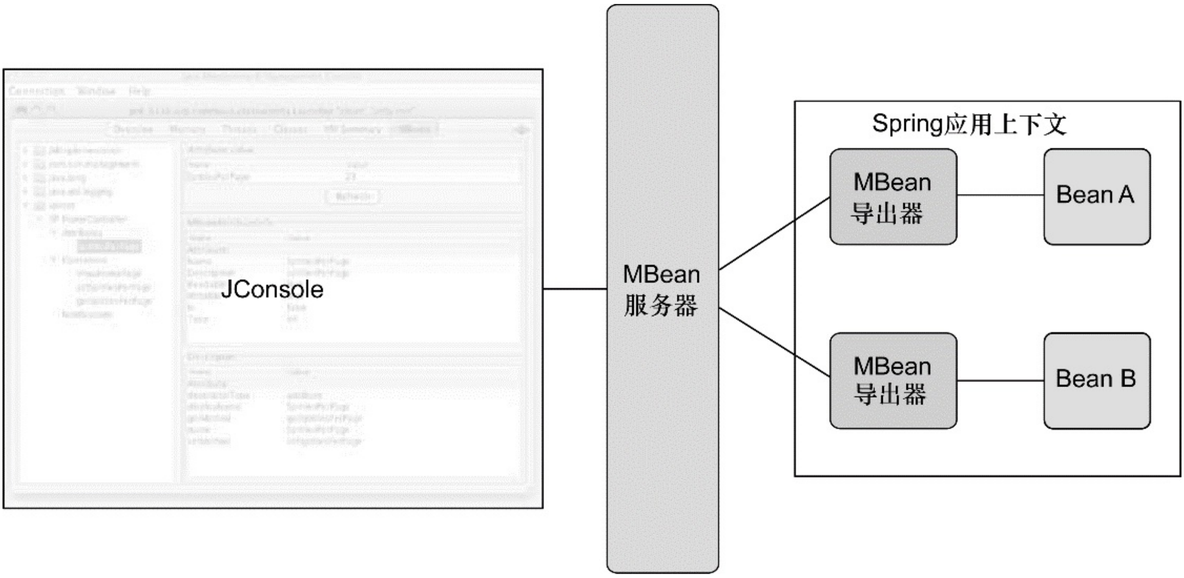

图20.1　Spring的MBeanExporter可以将Spring bean的属性和方法导出为MBean服务器中的JMX属性和操作。通过JMX服务器，JMX管理工具（例如JConsole）可以查看到正在运行的应用程序的内部情况

下面的@Bean方法在Spring中声明了一个MBeanExporter，它会将spittleControllerbean导出为一个模型MBean：

```
@Bean
public MBeanExporter mbeanExporter(SpittleController spittleController) {
  MBeanExporter exporter = new MBeanExporter();
  Map<String, Object> beans = new HashMap<String, Object>();
  beans.put("spitter:name=SpittleController", spittleController);
  exporter.setBeans(beans);
  return exporter;
}
```

配置MBeanExporter的最简单方式是为它的beans属性配置一个Map集合，该集合中的元素是我们希望暴露为JMX MBean的一个或多个bean。每个Map条目的key就是MBean的名称（由管理域的名字和一个key-value对组成，在SpittleController MBean示例中是spitter:name=HomeController），而Map条目的值则是需要暴露的Spring bean引用。在这里，我们将输出spittleControllerbean，以便它的属性可以通过JMX在运行时进行管理。

通过MBeanExporter，spittleControllerbean将作为模型MBean以SpittleController的名称导出到MBean服务器中，以实现管理功能。图20.2展示了通过JConsole查看SpittleControllerMBean时的情况。

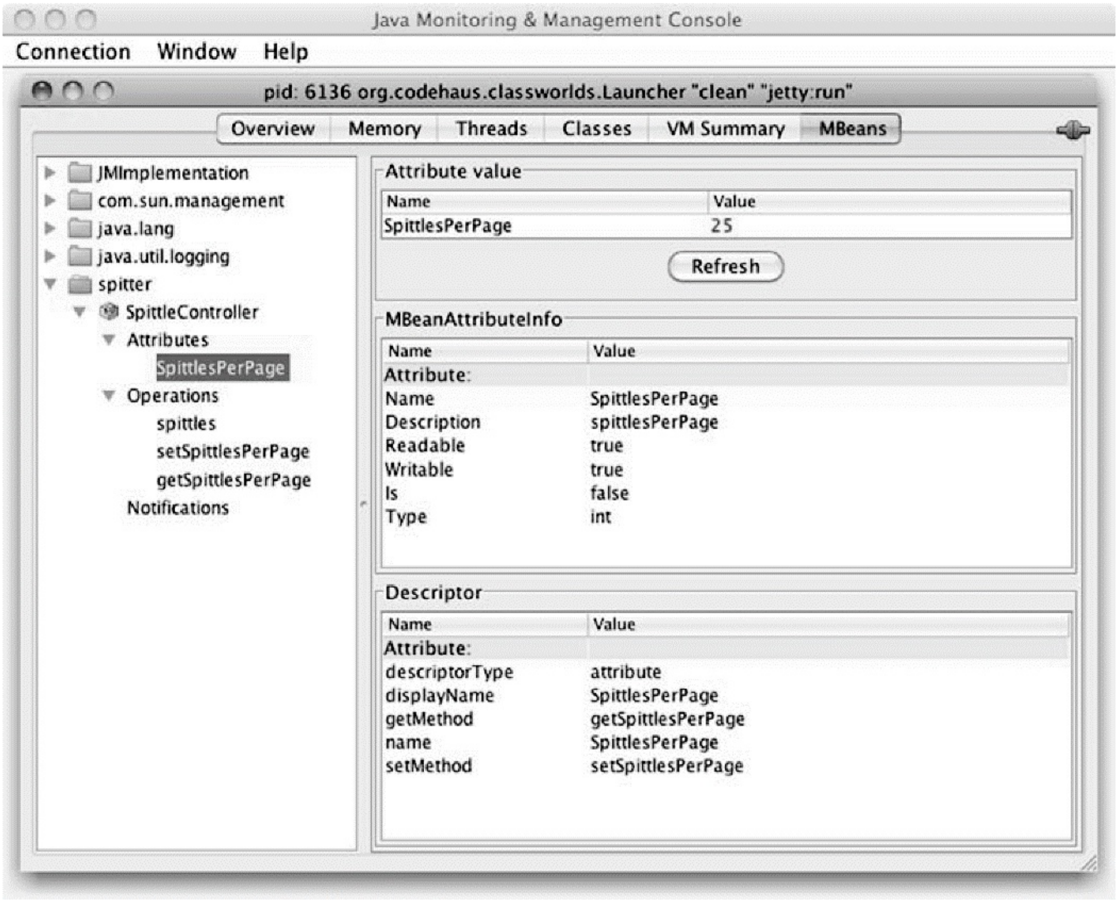

图20.2　SpittleController导出为MBean，并且可以通过JConsole查看

如图20.2的左侧所示，`SpittleController`所有的public成员都被导出为`MBean`的操作或属性。这可能并不是我们所希望看到的结果，我们真正需要的只是可以配置`spittlesPerPage`属性。我们不需要调用`spittles()`方法或`SpittleController`中的其他方法或属性。因此，我们需要一个方式来筛选所需要的属性或方法。

为了对MBean的属性和操作获得更细粒度的控制，Spring提供了几种选择，包括：

- 通过名称来声明需要暴露或忽略的bean方法；
- 通过为bean增加接口来选择要暴露的方法；
- 通过注解标注bean来标识托管的属性和操作。

我们会尝试每一种方式来决定哪一种最适合`SpittleController`MBean。我们首先通过名称来选择bean的哪些方法需要暴露。

MBean服务器从何而来

根据以上配置，MBeanExporter会假设它正在一个应用服务器中（例如Tomcat）或提供MBean服务器的其他上下文中运行。但是，如果Spring应用程序是独立的应用或运行的容器没有提供MBean服务器，我们就需要在Spring上下文中配置一个MBean服务器。

在XML配置中，<context:mbean-server>元素可以为我们实现该功能。如果使用Java配置的话，我们需要更直接的方式，也就是配置类型为MBeanServerFactoryBean的bean（这也是在XML中<context:mbean-server>元素所作的事情）。

MBeanServerFactoryBean会创建一个MBean服务器，并将其作为Spring应用上下文中的bean。默认情况下，这个bean的ID是mbeanServer。了解到这一点，我们就可以将它装配到MBeanExporter的server属性中用来指定MBean要暴露到哪个MBean服务器中。

20.1.1　通过名称暴露方法

MBean信息装配器（MBean info assembler）是限制哪些方法和属性将在MBean上暴露的关键。其中有一个MBean信息装配器是MethodNameBasedMBean-InfoAssembler。这个装配器指定了需要暴露为MBean操作的方法名称列表。对于SpittleController bean来说，我们希望把spittlePerPage暴露为托管属性。基于方法名的装配器如何帮我们导出一个托管属性呢？

我们回顾下JavaBean的规则（这不是Spring Bean所必需的），spittlesPerPage属性需要定义对应的存取器（accessor）方法，方法名必须为setSpittlesPerPage()和getSpittlesPerPage()。为了限制MBean所暴露的内容，我们需要告诉MethodNameBaseMBeanInfoAssembler仅在MBean的接口中包含这两个方法。如下MethodNameBaseMBeanInfoAssembler的bean声明就配置了这些方法：

```
@Bean
public MethodNameBasedMBeanInfoAssembler assembler() {
  MethodNameBasedMBeanInfoAssembler assembler =
          new MethodNameBasedMBeanInfoAssembler();
  assembler.setManagedMethods(new String[] {
    "getSpittlesPerPage", "setSpittlesPerPage"
  });
  return assembler;
}
```

managedMethods属性可以接受一个方法名称的列表，指定了哪些方法将暴露为MBean的操作。因为本示例所配置的是spittlesPerPage属性的存取器方法，所以spittlesPerPage属性也自然成为了MBean的托管属性。

为了让这个装配器能够生效，我们需要将它装配进MBeanExporter中：

```
@Bean
public MBeanExporter mbeanExporter(
      SpittleController spittleController,
      MBeanInfoAssembler assembler) {
  MBeanExporter exporter = new MBeanExporter();
  Map<String, Object> beans = new HashMap<String, Object>();
  beans.put("spitter:name=SpittleController", spittleController);
  exporter.setBeans(beans);
  exporter.setAssembler(assembler);
  return exporter;
}
```

现在如果我们启动应用，SpittleController的spittlesPerPage将作为有效的MBean托管属性，而spittles()方法并不会暴露为MBean的托管操作。图20.3展示了通过JConsole查看SpittleController的情况。

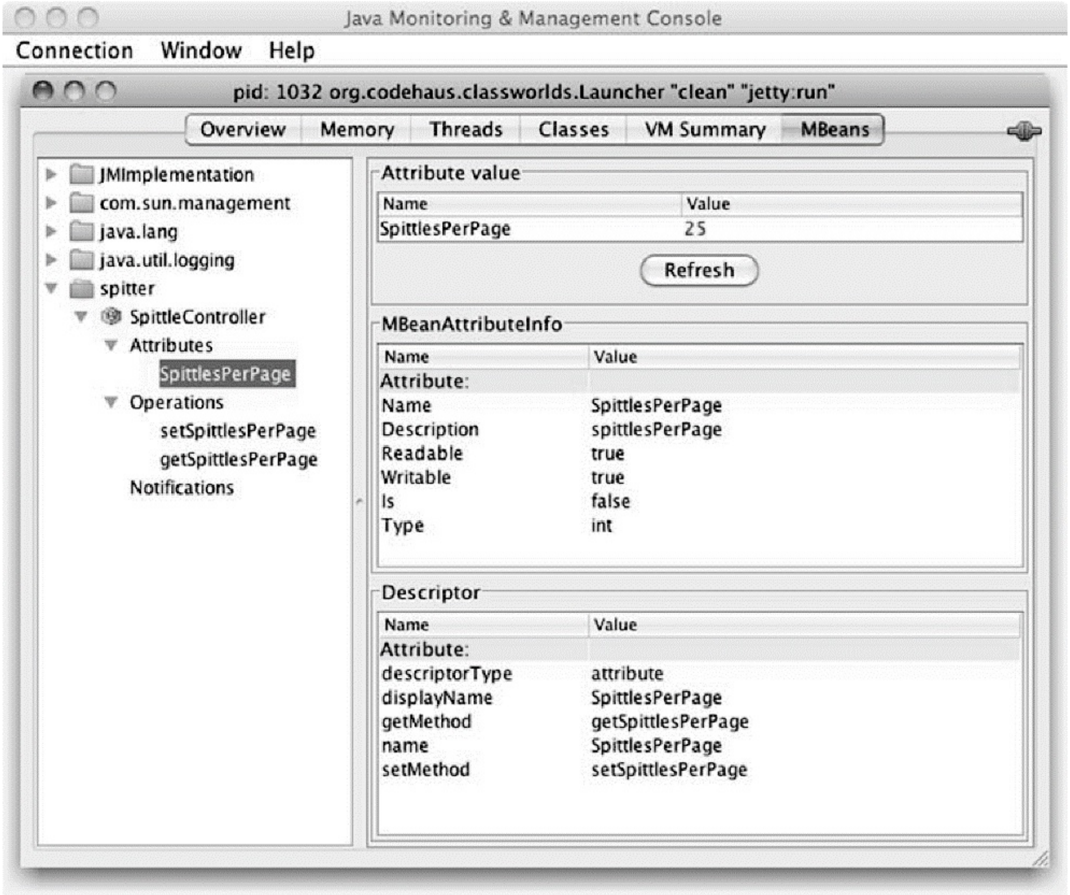

图20.3　当指定了哪些方法在SpittleController MBean上暴露后，spittles()方法不再作为MBean的托管操作

另一个基于方法名称的装配器是MethodExclusionMBeanInfoAssembler。这个MBean信息装配器是MethodNameBaseMBeanInfoAssembler的反操作。它不是指定哪些方法需要暴露为MBean的托管操作，MethodExclusionMBeanInfoAssembler指定了不需要暴露为MBean托管操作的方法名称列表。例如，在这里我们使用MethodExclusionMBeanInfoAssemble指定spittles()作为不暴露的方法：

```
@Bean
public MethodExclusionMBeanInfoAssembler assembler() {
  MethodExclusionMBeanInfoAssembler assembler =
          new MethodExclusionMBeanInfoAssembler();
  assembler.setIgnoredMethods(new String[] {
      "spittles"
  });
  return assembler;
}
```

基于方法名称的装配器是最直接和易于使用的。但是如果需要把多个Spring bean导出为MBean，我们能想象将出现什么样的情形吗？为装配器所配置的方法名称清单将会变得非常庞大；而且还有一种可能，我们希望暴露一个bean的某个方法，但不希望暴露另一个bean的同名方法。

很明显，在Spring配置方面，当导出多个MBean时，基于方法名称的方式并不能很好地满足此场景。让我们看一下如果使用接口暴露MBean的操作和属性是否更为合适。

20.1.2　使用接口定义MBean的操作和属性

Spring的InterfaceBasedMBeanInfoAssembler是另一种MBean信息装配器，可以让我们通过使用接口来选择bean的哪些方法需要暴露为MBean的托管操作。InterfaceBasedMBeanInfoAssembler与基于方法名称的装配器很相似，只不过不再通过罗列方法名称来确定暴露哪些方法，而是通过列出接口来声明哪些方法需要暴露。

例如，假设我们定义了一个名为SpittleControllerManagedOperations的接口，如下所示：

```
package com.habuma.spittr.jmx;

public interface SpittleControllerManagedOperations {
  int getSpittlesPerPage();
  void setSpittlesPerPage(int spittlesPerPage);
}
```

在这里，我们选择了setSpittlesPerPage()方法和getSpittlesPerPage()方法作为需要暴露的方法。再次提醒，这一对存取器方法间接暴露了spittlesPerPage属性作为MBean的托管属性。为了应用此装配器，我们只需要使用如下的assemblerbean替换之前基于方法名称的装配器即可：

```
@Bean
public InterfaceBasedMBeanInfoAssembler assembler() {
  InterfaceBasedMBeanInfoAssembler assembler =
          new InterfaceBasedMBeanInfoAssembler();
  assembler.setManagedInterfaces(
      new Class<?>[] { SpittleControllerManagedOperations.class }
  );
  return assembler;
}
```

managedInterfaces属性接受一个或多个接口组成的列表作为MBean的管理接口——在本示例中为SpittleControllerManagedOperations接口。

SpittleController并没有显式实现SpittleControllerManagedOperations接口，这可能并不明显，但相当有趣。这个接口只是为了标识导出的内容，但我们并不需要在代码中直接实现该接口。不过，SpittleController应该实现这个接口，其实也没有其他的原因，只是在MBean和实现类之间应该有一个一致的协议。

如果通过接口来选择MBean操作的话，最吸引人的一点在于我们可以把很多方法放在少量的接口中，从而确保InterfaceBasedMBeanInfoAssembler的配置尽量简洁。在输出多个MBean时，基于接口的方式可以帮助保持Spring配置的简洁。

最终，这些托管操作必须在某处声明，无论是在Spring配置中还是在某个接口中。此外，从代码角度看，托管操作的声明是一种重复——在接口中或Spring上下文中声明的方法名称与实现中所声明的方法名称存在重复。之所以存在这种重复，没有其他原因，仅仅是为了满足MBeanExporter的需要而产生的。

Java注解的一项工作就是帮助消除这种重复。让我们看看如何通过使用注解标注Spring管理的bean，从而将其导出MBean。

20.1.3 使用注解驱动的MBean

除了我向你展示的MBean信息装配器，Spring还提供了另一种装配器——Metadata-MBeanInfoAssembler，这种装配器可以使用注解标识哪些bean的方法需要暴露为MBean的托管操作和属性。我完全可以向你展示如何使用这种装配器，但我不会这么做。这是因为手工装配它非常繁杂，仅仅是为了使用注解并不值得这么做。相反，我将向你展示如何使用Spring context配置命名空间中的<context:mbean-export>元素。这个便捷的元素装配了MBean导出器以及为了在Spring启用注解驱动的MBean所需要的装配器。我们所需要做的就是使用它来替换我们之前所使用的MBeanExporterbean:

```
<context:mbean-export server="mbeanServer" />
```

现在，要把任意一个Spring bean转变为MBean，我们所需要做的仅仅是使用@ManagedResource注解标注bean并使用@ManagedOperation或@ManagedAttribute注解标注bean的方法。例如，如下的程序清单展示了如何使用注解把SpittleController导出为MBean。

程序清单**20.1** 通过注解把**HomeController**转变为**MBean**

```
package com.habuma.spittr.mvc;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spittr.service.SpittrService;
@Controller
@ManagedResource(objectName="spitter:name=SpittleController") //   <-- 将 SpittleController 导出为 MBean
public class SpittleController {
  ...
  @ManagedAttribute   //   <-- 将 spittlesPerPage 暴露为托管属性
  public void setSpittlesPerPage(int spittlesPerPage) {
    this.spittlesPerPage = spittlesPerPage;
  }

  @ManagedAttribute   //
  public int getSpittlesPerPage() {
    return spittlesPerPage;
  }
}
```

在类级别使用了@ManagedResource注解来标识这个bean应该被导出为MBean。objectName属性标识了域（Spitter）和MBean的名称（SpittleController）。

spittlesPerPage属性的存取器方法都使用了@ManagedAttribute注解来进行标注，这表示该属性应该暴露为MBean的托管属性。注意，其实并不需要使用注解同时标注这两个存取器方法。如果我们选择仅标注setSpittlesPerPage()方法，那我们仍可以通过JMX设置该属性，但这样的话我们将不能查看该属性的值。相反，如果仅仅标注getSpittlesPerPage()方法，那我们可以通过JMX查看该属性的值，但无法修改该属性的值。

同样需要提醒一下，我们还可以使用@ManagedOperation注解替换@ManagedAttribute注解来标注存取器方法。如下所示：

```
@ManagedOperation
public void setSpittlesPerPage(int spittlesPerPage) {
  this.spittlesPerPage = spittlesPerPage;
}

@ManagedOperation
public int getSpittlesPerPage() {
  return spittlesPerPage;
}
```

这会将方法暴露为MBean的托管操作，但是并不会把spittlesPerPage属性暴露为MBean的托管属性。这是因为在暴露MBean功能时，使用@ManagedOperation注解标注方法是严格限制方法的，并不会把它作为JavaBean的存取器方法。因此，使用@ManagedOperation可以用来把bean的方法暴露为MBean托管操作，而使用@ManagedAttribute可以把bean的属性暴露为MBean托管属性。

20.1.4 处理MBean冲突

到目前为止，我们已经看到可以使用多种方式在MBean服务器中注册MBean。在所有的示例中，我们为MBean指定的对象名称是由管理域名和key-value对组成的。如果MBean服务器中不存在与我们MBean名字相同的已注册的MBean，那我们的MBean注册时就不会有任何问题。但是如果名字冲突时，将会发生什么呢？

默认情况下，MBeanExporter将抛出InstanceAlreadyExistsException异常，该异常表明MBean服务器中已经存在相同名字的MBean。不过，我们可以通过MBeanExporter的registrationBehaviorName属性或者<context:mbean-export>的registration属性指定冲突处理机制来改变默认行为。

Spring提供了3种借助registrationBehaviorName属性来处理MBean名字冲突的机制：

- FAIL_ON_EXISTING：如果已存在相同名字的MBean，则失败（默认行为）；
- IGNORE_EXISTING：忽略冲突，同时也不注册新的MBean；
- REPLACING_EXISTING：用新的MBean覆盖已存在的MBean；

例如，如果我们使用MBeanExporter，我们可以通过设置registration-BehaviorName属性为RegistrationPolicy.IGNORE_EXISTING来忽略冲突，如下所示：

```
@Bean
public MBeanExporter mbeanExporter(
      SpittleController spittleController,
      MBeanInfoAssembler assembler) {
  MBeanExporter exporter = new MBeanExporter();
  Map<String, Object> beans = new HashMap<String, Object>();
  beans.put("spitter:name=SpittleController", spittleController);
  exporter.setBeans(beans);
  exporter.setAssembler(assembler);
  exporter.setRegistrationPolicy(RegistrationPolicy.IGNORE_EXISTING);
  return exporter;
}
```

registrationBehaviorName属性可以接受RegistrationPolicy中所定义的枚举值，每一个取值分别对应3种冲突处理机制的一种。

现在我们已使用MBeanExporter注册了我们的MBean，我们还需要一种方式来访问它们并进行管理。正如之前所看到的，我们可以使用诸如JConsole之类的工具来访问本地的MBean服务器，进而显示和操纵MBean，但是像JConsole之类的工具并不适合在程序中对MBean进行管理。我们如何在一个应用中操纵另一个应用中的MBean呢？幸运的是，还存在另一种方式可以把MBean作为远程对象进行访问。让我们进一步研究Spring对远程MBean的支持，了解如何通过远程接口以标准的方式来访问MBean。

20.2　远程MBean

虽然最初的JMX规范提及了通过MBean进行应用的远程管理，但是它并没有定义实际的远程访问协议或API。因此，会由JMX供应商定义自己的JMX远程访问解决方案，但这通常又是专有的。

为了满足以标准方式进行远程访问JMX的需求，JCP（Java Community Process）制订了JSR-160：Java管理扩展远程访问API规范（Java Management Extensions Remote API Specification）。该规范定义了JMX远程访问的标准，该标准至少需要绑定RMI和可选的JMX消息协议（JMX Messaging Protocol，JMXMP）。

在本小节中，我们将看到Spring如何远程访问MBean。我们首先从配置Spring把`SpittleController`导出为远程MBean开始，然后我们再了解如何使用Spring远程操纵MBean。

20.2.1 暴露远程MBean

使MBean成为远程对象的最简单方式是配置Spring的`ConnectorServer-FactoryBean`：

```
@Bean
public ConnectorServerFactoryBean connectorServerFactoryBean() {
  return new ConnectorServerFactoryBean();
}
```

`ConnectorServerFactoryBean`会创建和启动`JSR-160 JMXConnectorServer`。默认情况下，服务器使用JMXMP协议并监听端口9875——因此，它将绑定“`service:jmx:jmxmp://localhost:9875`”。但是我们导出MBean的可选方案并不局限于JMXMP。

根据不同JMX的实现，我们有多种远程访问协议可供选择，包括远程方法调用（Remote Method Invocation，RMI）、SOAP、Hessian/Burlap和IIOP（Internet InterORB Protocol）。为MBean绑定不同的远程访问协议，我们仅需要设置`ConnectorServerFactoryBean`的`serviceUrl`属性。例如，如果我们想使用RMI远程访问MBean，我们可以像下面示例这样配置：

```
@Bean
public ConnectorServerFactoryBean connectorServerFactoryBean() {
  ConnectorServerFactoryBean csfb = new ConnectorServerFactoryBean();
  csfb.setServiceUrl(
      "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter");
  return csfb;
}
```

在这里，我们将`ConnectorServerFactoryBean`绑定到了一个RMI注册表，该注册表监听本机的1099端口。这意味着我们需要一个RMI注册表运行时，并监听该端口。我们可以回顾下第15章，`RmiServiceExporter`可以为我们自动启动一个RMI注册表。

但是，我们在本示例中不使用RmiServiceExporter，而是通过在Spring中声明RmiRegistryFactoryBean来启动一个RMI注册表，如下面的@Bean方法所示：

```
@Bean
public RmiRegistryFactoryBean rmiRegistryFB() {
  RmiRegistryFactoryBean rmiRegistryFB = new RmiRegistryFactoryBean();
  rmiRegistryFB.setPort(1099);
  return rmiRegistryFB;
}
```

没错！现在我们的MBean可以通过RMI进行远程访问了。但是如果没有人通过RMI访问MBean的话，那就不值得这么做。所以现在让我们把关注点转向JMX远程访问的客户端，看看如何在Spring中装配一个远程MBean到JMX客户端中。

20.2.2　访问远程MBean

要想访问远程MBean服务器，我们需要在Spring上下文中配置MbeanServer-ConnectionFactoryBean。下面的bean声明装配了一个MbeanServerConnection-FactoryBean，该bean用于访问我们在上一节中所创建的基于RMI的远程服务器。

```
@Bean
public MBeanServerConnectionFactoryBean connectionFactoryBean() {
  MBeanServerConnectionFactoryBean mbscfb =
          new MBeanServerConnectionFactoryBean();
  mbscfb.setServiceUrl(
      "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter");
  return mbscfb;
}
```

顾名思义，MBeanServerConnectionFactoryBean是一个可用于创建MbeanServer-Connection的工厂bean。由MBeanServerConnectionFactoryBean所生成的MBeanServerConnection实际上是作为远程MBean服务器的本地代理。它能够以MBeanServerConnection的形式注入到其他bean的属性中：

```
@Bean
public JmxClient jmxClient(MBeanServerConnection connection) {
  JmxClient jmxClient = new JmxClient();
  jmxClient.setMbeanServerConnection(connection);
  return jmxClient;
}
```

MBeanServerConnection提供了多种方法，我们可以使用这些方法查询远程MBean服务器并调用MBean服务器内所注册的MBean的方

法。例如，如果我们希望知道在远程MBean服务器中有多少已注册的MBean，可以用如下的代码片段打印这些信息：

```
int mbeanCount = mbeanServerConnection.getMBeanCount();
System.out.println("There are " + mbeanCount + " MBeans");
```

我们还可以使用queryNames()方法查询远程服务器中所有MBean的名称：

```
java.util.Set mbeanNames = mbeanServerConnection.queryNames(null, null);
```

传递给queryNames()方法的两个参数用于过滤查询结果。如果将两个参数都设置为null，输出结果为所有已注册的MBean的名称。

查询远程MBean服务器上bean的数量和名称虽然很有趣，不过并不能完成更多的工作。远程访问 MBean服务器的真正价值在于访问远程服务器上已注册MBean的属性以及调用它们的方法。

为了访问MBean属性，我们可以使用getAttribute()和setAttribute()方法。例如，为了获取MBean属性的值，我们可以按照下面的方法调用getAttribute()方法：

```
String cronExpression = mbeanServerConnection.getAttribute(
    new ObjectName("spitter:name=SpittleController"), "spittlesPerPage");
```

同样，我们可以使用setAttribute()方法改变MBean属性的值：

```
mbeanServerConnection.setAttribute(
    new ObjectName("spitter:name=SpittleController"),
    new Attribute("spittlesPerPage", 10));
```

如果希望调用MBean的操作，那我们需要使用invoke()方法。下面的内容描述了如何调用SpittleController MBean的setSpittlesPerPage()方法：

```
mbeanServerConnection.invoke(
    new ObjectName("spitter:name=SpittleController"),
    "setSpittlesPerPage",
    new Object[] { 100 },
    new String[] {"int"});
```

我们还可以使用MBeanServerConnection的方法对远程MBean做很多其他的事情。我把它作为一个任务留给你。不过，通过MBeanServerConnection对远程MBean进行方法调用和属性设置是一种很笨拙的方法。要想调用setSpittlesPerPage()这样一个简单的方法，我们需要创建一个ObjectName实例，并向invoke()

方法传递几个参数。它并不是直观的方法调用。为了更直接地调用方法，我们需要代理远程MBean。

20.2.3 代理MBean

Spring的`MBeanProxyFactoryBean`是一个代理工厂bean，像我们在第15章中所演示的远程代理工厂bean类似。在前面所介绍的内容中，它们会提供代理，用来访问远程的Spring受管bean，与之不同，`MBcanProxyFactoryBean`可以让我们可以直接访问远程的MBean（就如同配置在本地的其他bean一样）。图20.4展示了它的工作原理。

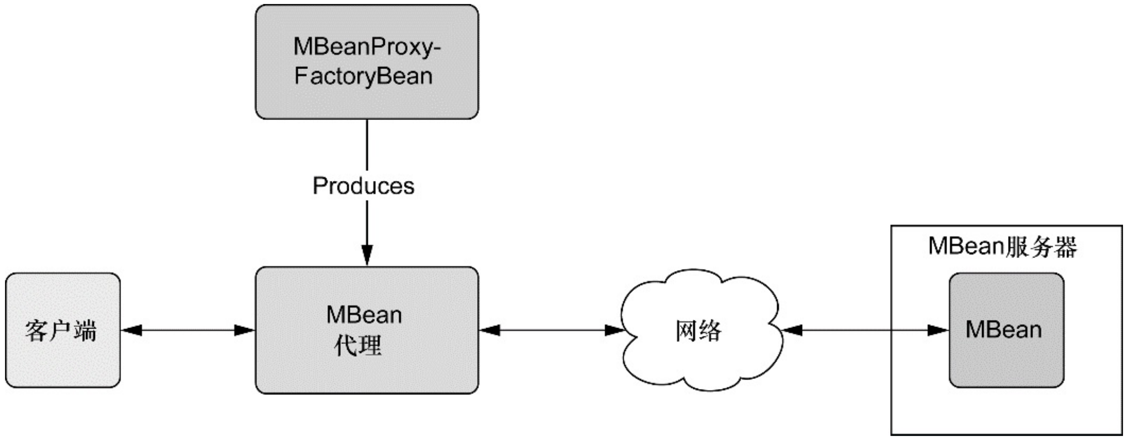

图20.4 MBeanFactoryBean创建远程MBean的代理。客户端通过此代理与远程MBean进行交互，就像它是本地Bean一样

例如，考虑如下的`MBeanProxyFactoryBean`声明：

```
@Bean
public MBeanProxyFactoryBean remoteSpittleControllerMBean(
        MBeanServerConnection mbeanServerClient) {
  MBeanProxyFactoryBean proxy = new MBeanProxyFactoryBean();
  proxy.setObjectName("");
  proxy.setServer(mbeanServerClient);
  proxy.setProxyInterface(SpittleControllerManagedOperations.class);
  return proxy;
}
```

`objectName`属性指定了远程MBean的对象名称。在这里是引用我们之前导出的`SpittleController`MBean。

`server`属性引用了`MBeanServerConnection`，通过它实现MBean所有通信的路由。在这里，我们注入了之前配置的`MBeanServerConnectionFactoryBean`。

最后，`proxyInterface`属性指定了代理需要实现的接口。在本示例中，我们使用20.1.2小节所定义的`SpittleControllerManagedOperations`接口。

对于上面声明的`remoteSpittleControllerMBean`，我们现在可以把它注入到类型为`SpittleControllerManagedOperations`的bean属性中，并使用它来访问远程的MBean。这样，我们就可以调用`setSpittlesPerPage()`和`getSpittlesPerPage()`方法了。

我们已经看到与MBean通信的几种方式，现在我们可以在应用运行的时候显示和调整Spring bean配置。但是目前为止，这都是单方面的会话。都是我们与MBean在沟通。现在是时候通过监听通知（notification）来倾听它们在说什么。

20.3　处理通知

通过查询MBean获得信息只是查看应用状态的一种方法。但当应用发生重要事件时，如果希望能够及时告知我们，这通常不是最有效的方法。

例如，假设Spittr应用保存了已发布的Spittle数量，而我们希望知道每发布一百万Spittle时的精确时间（例如一百万、两百万、三百万等）。一种解决方法是编写代码定期查询数据库，计算Spittle的数量。但是执行这种查询会让应用和数据库都很繁忙，因为它需要不断的检查Spittle的数量。

与重复查询数据库获得Spittle的数量相比，更好的方式是当这类事件发生时让MBean通知我们。JMX通知（JMX notification，如图20.5 所示）是MBean与外部世界主动通信的一种方法，而不是等待外部应用对MBean进行查询以获得信息。

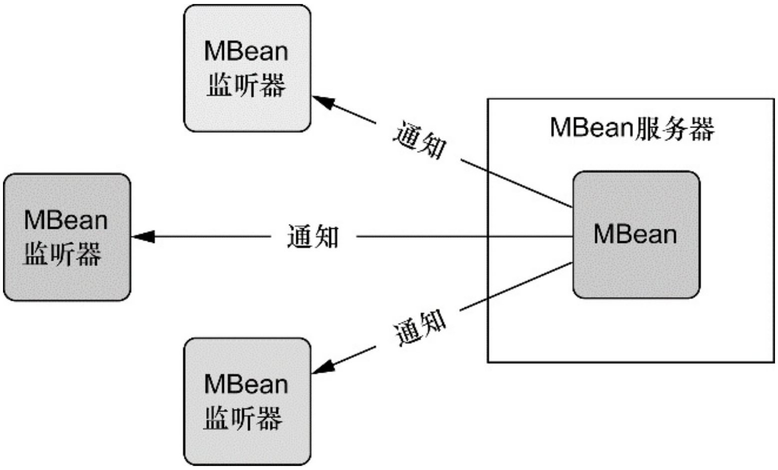

图20.5　JMX通知使MBean与外部世界进行主动通信

Spring通过`NotificationPublisherAware`接口提供了发送通知的支持。任何希望发送通知的MBean都必须实现这个接口。例如，请查看如下程序清单中的`SpittleNotifierImpl`。

程序清单**20.2**　使用**NotificationPublisher**来发送**JMX**通知

```
package com.habuma.spittr.jmx;
import javax.management.Notification;
import org.springframework.jmx.export.annotation.ManagedNotification;
import org.springframework.jmx.export.annotation.ManagedResource;
import
org.springframework.jmx.export.notification.NotificationPublisher;
import
org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.stereotype.Component;
@Component
@ManagedResource("spitter:name=SpitterNotifier")
@ManagedNotification(
        notificationTypes="SpittleNotifier.OneMillionSpittles",
        name="TODO")
public class SpittleNotifierImpl
    implements NotificationPublisherAware, SpittleNotifier {    <-- 实现 NotificationPublisherAware 接口

  private NotificationPublisher notificationPublisher;

  public void setNotificationPublisher(                         <-- 注入 notificationPublisher
          NotificationPublisher notificationPublisher) {
    this.notificationPublisher = notificationPublisher;
  }

  public void millionthSpittlePosted() {
    notificationPublisher.sendNotification(                     <-- 发送通知
            new Notification(
                    "SpittleNotifier.OneMillionSpittles", this, 0));
  }

}
```

正如我们所看到的，`SpittleNotifierImpl`实现了`NotificationPublisherAware`接口。这并不是一个要求苛刻的

接口，它仅要求实现一个方法：setNotificationPublisher。

SpittleNotificationImpl也实现了SpittleNotifier接口的方法：millionthSpittlePosted()。这个方法使用了setNotificationPublisher()方法所注入的NotificationPublisher来发送通知：我们的Spittle数量又到了一个新的百万级别。

一旦sendNotification()方法被调用，就会发出通知。嗯……好像我们还没决定谁来接收这个通知。那就让我们建立一个通知监听器来监听和处理通知。

20.3.1 监听通知

接收MBean通知的标准方法是实现javax.management.NotificationListener接口。例如，考虑一下PagingNotificationListener：

```
package com.habuma.spittr.jmx;
import javax.management.Notification;
import javax.management.NotificationListener;
public class PagingNotificationListener
       implements NotificationListener {

  public void handleNotification(
          Notification notification, Object handback) {
     // ...
  }

}
```

PagingNotificationListener是一个典型的JMX通知监听器。当接收到通知时，将会调用handleNotification()方法处理通知。大概的逻辑可能是，PagingNotification-Listener的handleNotification()方法将向寻呼机或手机上发送消息来告知Spittle数量又到了一个新的百万级别（我把实际的实现留给读者自己完成）。

剩下的工作只需要使用MBeanExporter注册PagingNotificationListener：

```
@Bean
public MBeanExporter mbeanExporter() {
  MBeanExporter exporter = new MBeanExporter();
  Map<?, NotificationListener> mappings =
          new HashMap<?, NotificationListener>();
  mappings.put("Spitter:name=PagingNotificationListener",
               new PagingNotificationListener());
  exporter.setNotificationListenerMappings(mappings);
  return exporter;
}
```

MBeanExporter的notificationListenerMappings属性用于在监听器和监听器所希望监听的MBean之间建立映射。在本示例中，我们建立了PagingNotificationListener来监听由SpittleNotifier MBean所发布的通知。

20.4　小结

JMX是对应用程序进行操纵的一扇窗口。在本章，我们了解了如何配置Spring自动地把Spring bean导出为JMX MBean，从而可以让我们通过JMX管理工具查看和操作bean的信息。我们也了解了当MBean和工具彼此距离很远时，如何创建和使用远程MBean。最后，我们还了解了如何使用Spring发布和监听JMX通知。

现在你或许注意到这本书剩余的页数越来越少，我们的Spring之旅即将结束。但是在这之前，我们沿途还会经停一站。在下一章，我们将会看一下Spring Boot，这是开发Spring应用的一种新方法，借助这种令人激动的新方法我们可以只保留很少的显式配置，甚至可能完全没有配置。

第21章　借助Spring Boot简化Spring开发

本章内容：

- 使用Spring Boot Starter添加项目依赖
- 自动化的 bean 配置
- Groovy 与 Spring Boot CLI
- Spring Boot Actuator

在我刚开始学习微积分课程的时候，我们学习了函数的导数。当时我们使用非常复杂的极限来计算函数的导数。即便函数非常简单，计算导数相关的工作依然像噩梦一样。

在布置完作业、建立完学习小组并考完试后，班上的大多数同学都能够完成这项任务了。但是它的单调无趣依然让我们无法忍受。如果"微积分（上）"的课程就这样的话，那在"微积分（下）"中，又该有怎样恐怖的数学计算在等着我们呢？

然后，老师给我们开了一个玩笑。通过使用一个简单的公式就能快速将导数计算出来（如果你学习过微积分的话，你应该能够明白我说的是什么）。通过这种新技巧，在以前计算一个函数导数的时间内，我们能够计算出十多个函数的导数。

此时，一位同学向老师提出了一个问题，这也是我们每位同学所想的："您为什么不在第一天就教会我们这个公式呢？！"

老师这样解释，比较困难的那种方法能够帮助我们理解导数的含义、告诉我们它的特性，并说这种方式对我们有这样那样的好处。

现在，我们用整本书的篇幅介绍了Spring，我发现自己处在类似于微积分老师那样的位置。尽管Spring带来的主要益处就是简化Java开发，但本章将会介绍Spring Boot如何让这项任务变得更加简单。从Spring创建以来，Spring Boot大概是Spring领域中最令人兴奋的事情了。它在Spring之上，构建了全新的开发模型，移除了开发Spring应用中很多单调乏味的内容。

我们首先整体上了解一下Spring Boot，看它是如何简化Spring的。在本章结束之前，我们将会使用Spring Boot构建一个完整的（尽管比较简单）应用程序。

21.1 Spring Boot简介

在Spring家族中，Spring Boot是令人兴奋（也许我敢说它是改变游戏规则的）的新项目。它提供了四个主要的特性，能够改变开发Spring应用程序的方式：

- Spring Boot Starter：它将常用的依赖分组进行了整合，将其合并到一个依赖中，这样就可以一次性添加到项目的Maven或Gradle构建中；
- 自动配置：Spring Boot的自动配置特性利用了Spring 4对条件化配置的支持，合理地推测应用所需的bean并自动化配置它们；
- 命令行接口（*Command-line interface*，*CLI*）：Spring Boot的CLI发挥了Groovy编程语言的优势，并结合自动配置进一步简化Spring应用的开发；
- Actuator：它为Spring Boot应用添加了一定的管理特性。

在本章中，我们将会使用Spring Boot的所有特性构建一个小型的应用程序。但首先，我们快速了解一下每项特性，更好地体验它们如何简化Spring编程模型。

21.1.1 添加Starter依赖

有两种烤制蛋糕的方式，有热情的人会将面粉、鸡蛋、糖、发酵粉、盐、奶油、香草调料以及牛奶混合在一起，和成糊状。或者也可以购买预先打包好的蛋糕，它包含了所需的大部分原料，我们只需添加一些含水分的材料即可，如水、鸡蛋和植物油。

预先打包好的蛋糕将制作蛋糕过程中所需的各种材料集合在了一起，作为一项材料来使用，与之类似，Spring Boot Starter将应用所需的各种依赖聚合成一项依赖。

为了阐述该功能，假设我们要从头开始编写一个新的Spring应用。这是一个Web项目，所以需要使用Spring MVC。同时，还要有REST API将资源暴露为JSON，所以在构建中需要包含Jackson JSON库。

因为应用需要使用JDBC从关系型数据库中存储和查询数据，因此我

们希望确保包含了Spring的JDBC模块（为了使用`JdbcTemplate`）和Spring的事务模块（为了使用声明式事务的支持）。对于数据库本身，H2数据库是个不错的选择。

对了，我们还需要使用Thymeleaf来建立Spring MVC视图。

如果使用Gradle构建项目的话，在build.gradle中（至少）需要包含如下的依赖：

```
dependencies {
  compile("org.springframework:spring-web:4.0.6.RELEASE")
  compile("org.springframework:spring-webmvc:4.0.6.RELEASE")
  compile("com.fasterxml.jackson.core:jackson-databind:2.2.2")
  compile("org.springframework:spring-jdbc:4.0.6.RELEASE")
  compile("org.springframework:spring-tx:4.0.6.RELEASE")
  compile("com.h2database:h2:1.3.174")
  compile("org.thymeleaf:thymeleaf-spring4:2.1.2.RELEASE")
}
```

幸好，Gradle能够非常简洁地表达依赖。（为简单起见，我不再展现这个依赖列表在Maven的pom.xml文件是什么样子了。）即便如此，创建这个文件还是牵扯到许多的事情，而对它的维护则会更加麻烦。这些依赖之间是如何协作的呢？当应用程序不断地成长和演进，依赖管理将会变得更加具有挑战性。

但是，如果我们使用Spring Boot Starer所提供的预打包依赖的话，那么Gradle依赖列表能够更加简短一些：

```
dependencies {
  compile("org.springframework.boot:spring-boot-starter-web:
          1.1.4.RELEASE")
  compile("org.springframework.boot:spring-boot-starter-jdbc:
          1.1.4.RELEASE")
  compile("com.h2database:h2:1.3.174")
  compile("org.thymeleaf:thymeleaf-spring4:2.1.2.RELEASE")
}
```

可以看到，Spring Boot的Web和JDBC Starter取代了几个更加细粒度的依赖。我们依然还需要包含H2和Thymeleaf的依赖，不过其他的依赖都已经放到了Starter中。除了依赖列表更加简短，我们可以相信由Starter所提供的依赖版本能够互相兼容。

Spring Boot提供了多个Starter，Web和JDBC只是其中的两个。表21.1列出了我在编写本章时，所有可用的Starter。

表**21.1　Spring Boot Starter**依赖将所需的常见依赖按组聚集在一起，形成单条依赖

Starter	所提供的依赖
spring-boot-starter-actuator	spring-boot-starter、spring-boot-actuator、spring-core
spring-boot-starter-amqp	spring-boot-starter、spring-boot-rabbit、spring-core、spring-tx
spring-boot-starter-aop	spring-boot-starter、spring-aop、AspectJ Runtime、AspectJ Weaver、spring-core
spring-boot-starter-batch	spring-boot-starter、HSQLDB、spring-jdbc、spring-batch-core、spring-core
spring-boot-starter-elasticsearch	spring-boot-starter、 spring-data-elasticsearch、 spring-core、 spring-tx
spring-boot-starter-gemfire	spring-boot-starter、 Gemfire、 spring-core、 spring-tx、 spring-context、 spring-context-support、 spring-data-gemfire
spring-boot-starter-data-jpa	spring-boot-starter、 spring-boot-starter-jdbc、 spring-boot-starter-aop、 spring-core、 Hibernate EntityManager、 spring-orm、 spring-data-jpa、 spring-aspects
spring-boot-starter-data-mongodb	spring-boot-starter、 MongoDB Java 驱动、 spring-core、 spring-tx、 spring-data-mongodb
spring-boot-starter-data-rest	spring-boot-starter、 spring-boot-starter-web、 Jackson 注解、 Jackson 数据绑定、 spring-core、 spring-tx、 spring-data-rest-webmvc
spring-boot-starter-data-solr	spring-boot-starter、 Solrj、 spring-core、 spring-tx、 spring-data-solr、 Apache HTTP Mime
spring-boot-starter-freemarker	spring-boot-starter、 spring-boot-starter-web、 Freemarker、 spring-core、 spring-context-support

spring-boot-starter-groovy-templ-ates	spring-boot-starter、 spring-boot-starter-web、 Groovy、 Groovy 模板、spring-core
spring-boot-starter-hornetq	spring-boot-starter、 spring-core、 spring-jms、 Hornet JMS Client
spring-boot-starter-integration	spring-boot-starter、 spring-aop、 spring-tx、 spring-web、 spring-webmvc、 spring-integration-core、 spring-integration-file、 spring-integration-http、 spring-integration-ip、 spring-integration-stream
spring-boot-starter-jdbc	spring-boot-starter、 spring-jdbc 、tomcat-jdbc、 spring-tx
spring-boot-starter-jetty	jetty-webapp、 jetty-jsp
spring-boot-starter-log4j	jcl-over-slf4j、 jul-to-slf4j 、slf4j-log4j12、log4j
spring-boot-starter -logging	jcl-over-slf4j、 jul-to-slf4j 、log4j-over-slf4j、 logback-classic
spring-boot-starter-mobile	spring-boot-starter、 spring-boot-starter-web、 spring-mobile-device
spring-boot-starter-redis	spring-boot-starter、 spring-data-redis、 lettuce
spring-boot-starter-remote-shell	spring-boot-starter-actuator、 spring-context、 org.crashub.**
spring-boot-starter-security	spring-boot-starter、 spring-security-config、 spring-security-web、 spring-aop、 spring-beans、 spring-context、 spring-core、 spring-expression、 spring-web

spring-boot-starter-social-facebook	spring-boot-starter、 spring-boot-starter-web、 spring-core、 spring-social-config、 spring-social-core、 spring-social-web、 spring-social-facebook
spring-boot-starter-social-twitter	spring-boot-starter、 spring-boot-starter-web、 spring-core、 spring-social-config、 spring-social-core、 spring-social-web、 spring-social-twitter
spring-boot-starter-social-linkedin	spring-boot-starter、 spring-boot-starter-web、 spring-core、 spring-social-config、 spring-social-core、 spring-social-web、 spring-social-linkedin
spring-boot-starter	spring-boot、 spring-boot-autoconfigure、 spring-boot-starter-logging
spring-boot-starter-test	spring-boot-starter-logging、 spring-boot、 junit、mockito-core、 hamcrest-library、 spring-test
spring-boot-starter-thymeleaf	spring-boot-starter、 spring-boot-starter-web、 spring-core、 thymeleaf-spring4、 thymeleaf-layout-dialect
spring-boot-starter-tomcat	tomcat-embed-core、 tomcat-embed-logging-juli
spring-boot-starter-web	spring-boot-starter、 spring-boot-starter-tomcat、 jackson-databind、 spring-web、 spring-webmvc
spring-boot-starter-websocket	spring-boot-starter-web、 spring-websocket、 tomcat-embed-core、 tomcat-embed-logging-juli
spring-boot-starter-ws	spring-boot-starter、 spring-boot-starter-web、 spring-core、 spring-jms、 spring-oxm、 spring-ws-core、 spring-ws-support

如果查看这些Starter依赖的内部原理，你会发现Starter的工作方式也没有什么神秘之处。它使用了Maven和Gradle的依赖传递方案，Starter

在自己的pom.xml文件中声明了多个依赖。当我们将某一个Starter依赖添加到Maven或Gradle构建中的时候，Starter的依赖将会自动地传递性解析。这些依赖本身可能也会有其他的依赖。一个Starter可能会传递性地引入几十个依赖。

需要注意，很多Starter引用了其他的Starter。例如，mobile Starter就引用了Web Starter，而后者又引用了Tomcat Starter。大多数的Starter都会引用`spring-boot-starter`，它实际上是一个基础的Starter（当然，它也依赖了logging Starter）。依赖是传递性的，将mobile Starter添加为依赖之后，就相当于添加了它下面的所有Starter。

21.1.2 自动配置

Spring Boot的Starter减少了构建中依赖列表的长度，而Spring Boot的自动配置功能则削减了Spring配置的数量。它在实现时，会考虑应用中的其他因素并推断你所需要的Spring配置。

作为样例，让我们重新回忆第6章（程序清单6.4），要将Thymeleaf模板作为Spring MVC的视图，至少需要三个bean：`ThymeleafViewResolver`、`SpringTemplateEngine`和`TemplateResolver`。但是，使用Spring Boot自动配置的话，我们需要做的仅仅是将Thymeleaf添加到项目的类路径中。如果Spring Boot探测到Thymeleaf位于类路径中，它就会推断我们需要使用Thymeleaf实现Spring MVC的视图功能，并自动配置这些bean。

Spring Boot Starter也会触发自动配置。例如，在Spring Boot应用中，如果我们想要使用Spring MVC的话，所需要做的仅仅是将Web Starter作为依赖放到构建之中。将Web Starter作为依赖放到构建中以后，它会自动添加Spring MVC依赖。如果Spring Boot的Web自动配置探测到Spring MVC位于类路径下，它将会自动配置支持Spring MVC的多个bean，包括视图解析器、资源处理器以及消息转换器（等等）。我们接下来需要做的就是编写处理请求的控制器。

21.1.3 Spring Boot CLI

Spring Boot CLI充分利用了Spring Boot Starter和自动配置的魔力，并添加了一些Groovy的功能。它简化了Spring的开发流程，通过CLI，我们能够运行一个或多个Groovy脚本，并查看它是如何运行的。在应用的运行过程中，CLI能够自动导入Spring类型并解析依赖。

用来阐述Spring Boot CLI的最有趣的例子就是如下的Groovy脚本：

```
@RestController
class Hi {
  @RequestMapping("/")
  String hi() {
        "Hi!"
  }
}
```

不管你是否相信，这是一个完整的（尽管比较简单）Spring应用，它可以在Spring Boot CLI中运行。包括空格，它的长度只有82个字符。你可以将其粘贴到Twitter客户端，并分享给你的朋友们。

去掉不必要的空格，我们能够得到只有64个字符的一行代码：

```
@RestController class Hi{@RequestMapping("/")String hi(){"Hi!"}}
```

这个版本更加简单，在一条Twitter的推文中，我们可以粘贴两次。但它依然是一个完整可运行的（尽管特性比较简陋）Spring应用。如果你已经安装过Spring Boot CLI，我们可以使用如下的命令行来运行它：

```
$ spring run Hi.groovy
```

以推文的形式来展示Spring Boot CLI的功能是很有意思的，但是它所能做的事情并不仅限于我们所看到的这些。在21.3小节中，我们将会看到如何使用Groovy和CLI构建更加完整的应用。

21.1.4 Actuator

Spring Boot Actuator为Spring Boot项目带来了很多有用的特性，包括：

- 管理端点；
- 合理的异常处理以及默认的“/error”映射端点；
- 获取应用信息的“/info”端点；
- 当启用Spring Security时，会有一个审计事件框架。

这些特性都是很有用的，但Actuator最有用和最有意思的特性是管理端点。在21.4小节中，我们将会看到Spring Boot Actuator的几个样例，它开启了一扇窗，能够让我们洞悉应用的内部运行状况。

现在，我们对Spring Boot的四个主要特性已经有了基本的了解，接下

来我们将使用它们构建一个微小但完整的应用程序。

21.2 使用Spring Boot构建应用

在本章剩余的内容中，我将会向你展现如何使用Spring Boot构建完整且符合现实要求（real-world）的应用程序。当然，“符合现实要求”的定义标准会有些争议，对它的讨论超出了本章的范围。因此，与其在这里说构建符合现实要求的应用，还不如后退一步，说成我们所构建的应用程序比现实要求稍差一点，但是它能够代表使用Spring Boot所构建的更大型应用。

我们的应用是一个简单的联系人列表。它允许用户输入联系人信息（名字、电话号码以及Email地址），并且能够列出用户之前输入的所有联系人信息。

你可以自由选择使用Maven还是Gradle来构建应用程序，我更喜欢Gradle，但是如果你喜欢Maven的话，我也将会列出所需的Maven代码。如下的程序清单展现了起始的build.gradle文件。开始的时候，依赖部分是空的，但是在这个过程中，我们将会使用依赖填充这部分的内容。

程序清单**21.1** **Contacts**应用所需的**Gradle**构建文件

```
buildscript {
  repositories {
    mavenLocal()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:
             1.1.4.RELEASE")
  }
}

apply plugin: 'java'                              使用
apply plugin: 'spring-boot'                    <─ Spring Boot 插件

jar {                        <── 构建 JAR 文件
  baseName = 'contacts'
  version =  '0.1.0'
}

repositories {
  mavenCentral()
}

dependencies {             <── 依赖将会放到这里

}

task wrapper(type: Wrapper) {
  gradleVersion = '1.8'
}
```

注意，构建中包含对Spring Boot Gradle的`buildscript`依赖。稍后将会看到，这会帮助我们生成一个可执行的超级JAR文件（uber-JAR），这个文件中将会包含应用的所有依赖。

如果你更喜欢Maven的话，如下的程序清单展现了完整的pom.xml文件。

程序清单**21.2　Contacts**应用所需的**Maven**构建文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.habuma</groupId>
  <artifactId>contacts</artifactId>
  <version>0.1.0</version>
  <packaging>jar</packaging>                    <-- 构建 JAR 文件

  <parent>
    <groupId>org.springframework.boot</groupId>              <-- 继承自
    <artifactId>spring-boot-starter-parent</artifactId>          Spring Boot
                                                                starter parent

    <version>1.1.4.RELEASE</version>
  </parent>

  <dependencies>                   <-- 依赖将会放到这里

  </dependencies>

  <build>
    <plugins>
      <plugin>                                          <-- 构建 JAR 文件
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>
```

与Gradle构建类似，这个Maven的pom.xml文件使用了Spring Boot Maven插件。这个Maven中的插件对应于Gradle插件，能够生成可执行的超级JAR文件。

同样需要注意的是，与Gradle构建不同，Maven构架有一个parent项目。我们让项目的Maven构建基于Spring Boot starter parent，这样的话，我们就能受益于Maven的依赖管理功能，对于项目中的很多依赖，就没有必要明确声明版本号了，因为版本号会从parent中继承得到。

按照Maven和Gradle项目的标准结构，完成后项目将会如下所示：

```
$ tree
.
├── build.gradle
├── pom.xml
└── src
    └── main
        ├── java
        │   └── contacts
        │       ├── Application.java
        │       ├── Contact.java
        │       ├── ContactController.java
        │       └── ContactRepository.java
        └── resources
            ├── schema.sql
            ├── static
            │   └── style.css
            └── templates
                └── home.html
```

不要担心现在缺失Java和其他的资源文件。在开发Contacts应用的过程中，我们将会在下面的几个小节中创建这些文件，首先将会从构建应用的Web层开始。

21.2.1 处理请求

因为我们要使用Spring MVC来开发应用的Web层，因此需要将Spring MVC作为依赖添加到构建中。我们已经讨论过，Spring Boot的Web Starter能够将Spring MVC需要的所有内容一站式添加到构建中。如下是我们所需的Gradle依赖：

```
compile("org.springframework.boot:spring-boot-starter-web")
```

如果使用Maven来进行构建的话，那么依赖将会如下所示：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

注意，因为Spring Boot parent项目已经指定了Web Starter依赖的版本，因此在项目的build.gradle和pom.xml文件中没有必要再显式指定版本信息。

Web Starter依赖就绪之后，使用Spring MVC需要的所有依赖都会添加到项目中。现在，我们就可以编写应用所需的控制器类了。

控制器相对会非常简单，包含展现联系人表单的HTTP `GET`请求以及处理表单提交的`POST`请求。它本身并没有做太多的事情，而是委托`ContactRepository`（稍后就会创建它）来持久化联系人信

息。程序清单21.3中的ContactController就能满足这些需求。

程序清单21.3　**ContactController**为**Contacts**应用处理基本的**Web**请求

```
package contacts;
import java.util.List;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/")
public class ContactController {
  private ContactRepository contactRepo;
                                                              注入
  @Autowired                                                  ContactRepositoy
  public ContactController(ContactRepository contactRepo) {  <-
    this.contactRepo = contactRepo;
  }

  @RequestMapping(method=RequestMethod.GET)               <-- 处理 GET "/"
   public String home(Map<String,Object> model) {
    List<Contact> contacts = contactRepo.findAll();
    model.put("contacts", contacts);
    return "home";
  }
  @RequestMapping(method=RequestMethod.POST)              <-- 处理 POST "/"
   public String submit(Contact contact) {
    contactRepo.save(contact);
    return "redirect:/";
  }
}
```

你首先可能会发现ContactController就是一个典型的Spring MVC控制器。尽管Spring Boot会管理构建依赖并最小化Spring配置，但是在编写应用逻辑的时候，编程模型是一致的。

在本例中，ContactController遵循了Spring MVC控制器的典型模式，它会展现表单并处理表单的提交。其中home()方法使用注入的ContactRepository来获取所有Contact对象的列表，并将它们放到模型中，然后把请求转交给home视图。这个视图将会展现联系人的列表以及添加新Contact的表单。submit()方法将会处理表单提交的POST请求，保存Contact，并重定向到首页。

因为ContactController使用了@Controller注解，所以组件扫描将会找到它。因此，我们不需要在Spring应用上下文中明确将其声明为bean。

而Contact模型类是一个简单的POJO，具有一些属性和存取器方法，如下面的程序清单所示。

程序清单**21.4** **Contact**是一个简单的领域类型

```
package contacts;

public class Contact {
  private Long id;                    <— 属性
  private String firstName;
  private String lastName;
  private String phoneNumber;
  private String emailAddress;

  public void setId(Long id) {              <— 存取器方法
    this.id = id;
  }

  public Long getId() {
    return id;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public String getLastName() {
    return lastName;
  }

  public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
  }

  public String getPhoneNumber() {
    return phoneNumber;
  }

  public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
  }

  public String getEmailAddress() {
    return emailAddress;
  }
}
```

应用程序的Web层基本上已经完成了，剩下的就是创建定义`home`视图的Thymeleaf模板。

21.2.2 创建视图

按照传统的方式，Java Web应用会使用JSP作为视图层的技术。但是，正如我在第6章所述，在这个领域有一个新的参与者。Thymeleaf的原生模板比JSP更加便于使用，而且它能够让我们以HTML的形式编写模板。鉴于此，我们将会使用Thymeleaf来定义Contacts应用的`home`视图。

首先，需要将Thymeleaf添加到项目的构建中。在本例中，我使用的是Spring 4，所以需要将Thymeleaf的Spring 4模块添加到构建之中。在Gradle中，依赖将会如下所示：

```
compile("org.thymeleaf:thymeleaf-spring4")
```

如果使用Maven的话，所需的依赖如下所示：

```
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf-spring4</artifactId>
</dependency>
```

需要记住的是，只要我们将Thymeleaf添加到项目的类路径下，就启用了Spring Boot的自动配置。当应用运行时，Spring Boot将会探测到类路径中的Thymeleaf，然后会自动配置视图解析器、模板解析器以及模板引擎，这些都是在Spring MVC中使用Thymeleaf所需要的。因此，在我们的应用中，不需要使用显式Spring配置的方式来定义Thymeleaf。

除了将Thymeleaf依赖添加到构建中，我们剩下所需要做的就是定义视图模板。程序清单21.5展现了home.html，这是定义home视图的Thymeleaf模板。

程序清单**21.5　home**视图渲染了一个创建新联系人的表单以及展现联系人的列表

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Spring Boot Contacts</title>
    <link rel="stylesheet" th:href="@{/style.css}" />     <—— 加载样式表
   </head>
  <body>
    <h2>Spring Boot Contacts</h2>

    <form method="POST">                                 <—— 新联系人的表单
       <label for="firstName">First Name:</label>
        <input type="text" name="firstName"></input><br/>
      <label for="lastName">Last Name:</label>
        <input type="text" name="lastName"></input><br/>
      <label for="phoneNumber">Phone #:</label>
        <input type="text" name="phoneNumber"></input><br/>
      <label for="emailAddress">Email:</label>
        <input type="text" name="emailAddress"></input><br/>
      <input type="submit"></input>
    </form>

    <ul th:each="contact : ${contacts}">        <—— 渲染联系人列表
       <li>
        <span th:text="${contact.firstName}">First</span>
      <span th:text="${contact.lastName}">Last</span> :
        <span th:text="${contact.phoneNumber}">phoneNumber</span>,
      <span th:text="${contact.emailAddress}">emailAddress</span>
    </li>
    </ul>
  </body>
</html>
```

它实际上是一个非常简单的Thymeleaf模板，分为两部分：一个表单和一个联系人的列表。表单将会`POST`数据到`ContactController`的`submit()`方法上，用来创建新的`Contact`。列表部分将会循环列出模型中的`Contact`对象。

为了使用这个模板，我们需要对其进行慎重地命名并放在项目的正确位置下。因为`ContactController`中`home()`方法所返回的逻辑视图名为`home`，因此模板文件应该命名为home.html，自动配置的模板解析器会在指定的目录下查找Thymeleaf模板，这个目录也就是相对于根类路径下的*templates*目录下，所以在Maven或Gradle项目中，我们需要将home.html放到“src/main/ resources/templates”中。

这个模板还有一点小事情需要处理，它所产生的HTML将会引用名为style.css的样式表。因此，需要将这个样式表放到项目中。

21.2.3　添加静态内容

正常来讲，在编写Spring应用时，我会尽量避免讨论样式和图片。当然，这些内容能够在很大程度上让各种应用（包括Spring应用）变得更加美观，令用户赏心悦目。但是，对于编写服务器端的Spring代码来说，这些静态内容就没有那么重要了。

但是，在Spring Boot中，有必要讨论一下它是如何处理静态内容的。当采用Spring Boot的Web自动配置来定义Spring MVC bean时，这些bean中会包含一个资源处理器（resource handler），它会将“/**”映射到几个资源路径中。这些资源路径包括（相对于类路径的根）：

- /META-INF/resources/
- /resources/
- /static/
- /public/

在传统的基于Maven/Gradle构建的项目中，我们通常会将静态内容放在“src/main/webapp”目录下，这样在构建所生成的WAR文件里面，这些内容就会位于WAR文件的根目录下。如果使用Spring Boot构建WAR文件的话，这依然是可选的方案。但是，我们也可以将静态内容放在资源处理器所映射的上述四个路径下。

所以，为了满足Thymeleaf模板对“/style.css”文件的引用，我们需要创建一个名为style.css文件，并将其放到如下所示的某一个位置中：

- /META-INF/resources/style.css
- /resources/style.css
- /static/style.css
- /public/style.css

具体的选择完全取决于你，我倾向于将静态内容放到“/public”中，不过这四个可选方案是等价的。

尽管style.css文件的内容与讨论无关，但是如下这个简单的样式表能够让应用看上去更加整洁：

```
body {
  background-color: #eeeeee;
  font-family: sans-serif;
}

label {
  display: inline-block;
  width: 120px;
  text-align: right;
}
```

不管你是否相信，对于这个简单的Contacts应用来说，我们已经完成了超过一半的任务！Web层全部完成了，接下来我们需要创建`ContactRepository`，用来处理`Contact`对象的持久化。

21.2.4　持久化数据

在Spring应用中，有多种使用数据库的方式。我们可以使用JPA或Hibernate将对象映射为关系型数据库中的表和列。或者，我们干脆放弃关系型数据库，使用其他类型的数据库，如Mongo或Neo4j。

对于Contacts应用来说，关系型数据库是不错的选择。我们将会使用H2数据库和JDBC（使用Spring的`JdbcTemplate`），让这个过程尽可能地简单。

选择这种方案就需要在构建中添加一些依赖。JDBC Starter依赖会将Spring `JdbcTemplate`需要的所有内容都引入进来。不过，要结合使用H2数据库的话，我们还需要添加H2依赖。如果使用Gradle的话，在`dependencies`代码块添加如下两行代码就能完成这项任务：

```
compile("org.springframework.boot:spring-boot-starter-jdbc")
compile("com.h2database:h2")
```

如果使用Maven构建的话，我们需要如下的两个`<dependency>`代码块：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

将这两项依赖添加到构建之中后，我们就可以编写Repository类了。如下程序清单中的`ContactRepository`将会使用注入的`JdbcTemplate`实现在数据库中读取和写入`Contact`对象。

程序清单**21.6　ContactRepository**能够从数据库中存取**Contact**

```
package contacts;
import java.util.List;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

@Repository
public class ContactRepository {
  private JdbcTemplate jdbc;

  @Autowired
  public ContactRepository(JdbcTemplate jdbc) {       <-- 注入 JdbcTemplate
     this.jdbc = jdbc;
  }

  public List<Contact> findAll() {                     查询联系人
    return jdbc.query(
       "select id, firstName, lastName, phoneNumber, emailAddress " +
      "from contacts order by lastName",
      new RowMapper<Contact>() {
        public Contact mapRow(ResultSet rs, int rowNum)
              throws SQLException {
          Contact contact = new Contact();
          contact.setId(rs.getLong(1));
          contact.setFirstName(rs.getString(2));
          contact.setLastName(rs.getString(3));
          contact.setPhoneNumber(rs.getString(4));
          contact.setEmailAddress(rs.getString(5));
          return contact;
        }
      });
  }

  public void save(Contact contact) {
    jdbc.update(                                      <-- 插入联系人
       "insert into contacts " +
        "(firstName, lastName, phoneNumber, emailAddress) " +
        "values (?, ?, ?, ?)",
      contact.getFirstName(), contact.getLastName(),
      contact.getPhoneNumber(), contact.getEmailAddress());
  }
}
```

与ContactController类似，这个Repository类非常简单。它与传统Spring应用中的Repository类并没有什么差别。从实现中，根本无法看出它要用于Spring Boot的应用程序中。findAll()方法使用注入的JdbcTemplate从数据库中获取Contact对象，save()方法使用注入的JdbcTemplate保存新的Contact对象。因为ContactRepository使用了@Repository注解，因此在组件扫描的时候，它会被发现并创建为Spring应用上下文中的bean。

但是，JdbcTemplate呢？我们难道不需要在Spring应用上下文中声明JdbcTemplatebean吗？为了声明它，我们是不是还要声明一个H2 DataSource？

对这两个问题的简短问答就是“不需要”。当Spring Boot探测到Spring

的JDBC模块和H2在类路径下的时候，自动配置就会发挥作用，将会自动配置JdbcTemplatebean和H2DataSourcebean。Spring Boot再一次为我们处理了所有的Spring配置。

那数据库模式该怎么处理呢？我们必须要自己来定义创建contacts表的模式，对不对？

这绝对是正确的！Spring Boot没有办法猜测contacts表会是什么样子。所以，我们需要定义模式，如下所示：

```
create table contacts (
  id identity,
  firstName varchar(30) not null,
  lastName varchar(50) not null,
  phoneNumber varchar(13),
  emailAddress varchar(30)
);
```

现在，我们只需要有一种方式加载这个“create table”的SQL并将其在H2数据库中执行就可以了。幸好，Spring Boot也涵盖了这项功能。如果我们将这个文件命名为schema.sql并将其放在类路径根下（也就是Maven或Gradle项目的“src/main/resources”目录下），当应用启动的时候，就会找到这个文件并进项数据加载。

21.2.5　尝试运行

Contacts应用非常简单，但是也算得上现实中的Spring应用。它具有Spring MVC控制器和Thymeleaf模板所定义的Web层，并且具有Repository和Spring JdbcTemplate所定义的持久层。

到此为止，我们已经编写完了Contacts所需的应用级别代码。不过，我们还没有编写任何形式的配置。我们没有编写任何Spring配置，也没有在web.xml或Servlet初始化类中配置DispatcherServlet。

如果我说不需要编写任何的配置，你会相信吗？

这应该做不到吧，毕竟在对Spring的批评中，人们都在说Spring全是配置，肯定有我们忽略掉的XML文件或Java配置类。我们所编写的Spring应用程序根本就不可能没有任何配置的……那么，我们到底能做到吗？

通常来讲，Spring Boot的自动配置特性消除了绝大部分或者全部的配置。因此，完全可能编写出没有任何配置的Spring应用程序。当然，

自动配置并不能涵盖所有的场景，因此典型的Spring Boot应用程序依然会需要一点配置。

具体到Contacts应用，我们不需要任何的配置。Spring的自动配置功能已经将所有的事情都做好了。

但是，我们需要有个特殊的类来启动Spring Boot应用。Spring本身并不知道自动配置的任何信息。程序清单21.7中的Application类就是Spring Boot启动类的典型例子。

程序清单**21.7**　初始化**Spring Boot**配置的简单启动类

```
package contacts;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan;

@ComponentScan
@EnableAutoConfiguration                    <-- 启用自动配置
 public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);    <-- 运行应用
   }
}
```

好吧，我承认`Application`中有那么一点配置。它使用`@ComponentScan`注解来启用组件扫描，另外它还使用了`@EnableAutoConfiguration`，这会启用Spring Boot的自动配置特性。但是，也就这么多了！除了这两行代码以外，Contacts再也没有什么配置了。

`Application`类最有意思的一点在于它具有一个`main()`方法。稍后将会看到，Spring Boot应用会以一种特殊的方法运行，正是这里的`main()`方法使这一切成为可能。在`main()`方法中，这行代码会告诉Spring Boot（通过`SpringApplication`类）根据`Application`中的配置以及命令行中的参数来运行。

现在，我们马上就可以运行应用了。剩下就是要进行构建。如果使用Gradle的话，那么如下的命令行会将项目构建到“build/libs/contacts-0.1.0.jar”中：

```
$ gradle build
```

如果你喜欢Maven的话，那么可以按照如下的方式构建项目：

```
$ mvn package
```

运行Maven构建后，你会在target文件夹下找到构建形成的结果。

现在，我们就可以运行它了。按照传统的方式，这意味着要将应用的WAR文件部署到Servlet容器中，如Tomcat或WebSphere。但是在这里，我们甚至没有WAR文件——构建形成的是一个JAR文件。

这没有什么问题。我们可以按照如下的方式从命令行运行它（引用的是基于Gradle构建的JAR文件）：

```
$ java -jar build/libs/contacts-0.1.0.jar
```

在几秒钟后，应用应该已经启动完成并且可以访问了。打开浏览器进入http://localhost:8080，你就应该可以输入联系人了。在输入几个联系人后，浏览器将会如图21.1所示。

你可能觉得这并不符合Web应用的运行方式。像这样从命令行运行应用非常简洁和方便，但是，对于你来讲，也许这并不理想。在你所工作的环境中，有可能需要将Web应用作为WAR文件部署到Web容器中。如果不提交WAR文件的话，可能不满足公司的部署策略。

图21.1　Spring Boot Contacts应用

好的，那也没有问题。

即便是对于生产环境，通过命令行来运行应用也是合理的方案，但是我理解你可能需要遵循公司的部署流程。这意味着需要构建和部署WAR文件。

好消息是，如果你需要WAR文件的话，并没有必要舍弃Spring Boot的简洁性。需要做的事情仅仅是稍微调整一下构建文件。在Gradle构建中，我们需要添加如下这行代码来应用“war”插件：

```
apply plugin: 'war'
```

除此之外，还需要将“jar”配置调整为“war”。这实际上就是将“j”替换为“w”：

```
war {
  baseName = 'contacts'
  version =  '0.1.0'
}
```

如果是Maven构建的项目，那会更加简单。只需将packaging从“jar”替换为“war”即可：

```
<packaging>war</packaging>
```

现在，我们可以重新构建项目，然后将会在构建目录中找到contacts-0.1.0.war文件。这个WAR文件文件可以部署到任意支持Servlet 3.0的容器中。另外，我们依然可以在命令行中运行这个应用：

```
$ java -jar build/libs/contacts-0.1.0.war
```

没错：这是一个可运行的WAR文件！对于两种场景来说，这都是最佳的方案。

我们可以看到，Spring Boot能够在很大程度上尽可能简化Spring应用的部署。Spring Boot Stater简化了项目构建的依赖，自动配置消除了显式的Spring配置。但稍后你会看到，如果再结合Groovy，它会更加简单。

21.3 组合使用Groovy与Spring Boot CLI

Groovy编程语言要比Java简单得多。它的语法允许有一些快捷方式，比如省略分号和`public`关键词。同时，Groovy类中的属性不像Java那样需要Setter和Getter方法。当然，Groovy还有其他的一些属性，能够消除Java代码中很多的繁文缛节。

如果你愿意使用Groovy编写应用代码并通过Spring Boot CLI运行的话，那么Spring Boot能够借助Groovy的简洁性进一步简化Spring应

用。为了阐述这一点，我们使用Groovy来重新编写Contacts应用程序。

为什么不呢？在初始版本的应用中，我们只有几个小的Java类，因此使用Groovy进行重写也没有太多的工作量。我们可以重用相同的Thymeleaf模板和schema.sql文件。既然我宣称Groovy能够进一步简化Spring，那重写应用也不是什么大事儿。

在这个过程中，我们还会移除一些代码。Spring Boot CLI本身就是启动器，所以不再需要前面所创建的`Application`类。Maven和Gradle构建文件也不再需要了，因为我们将会通过CLI运行未编译的Groovy文件。少了Maven和Gradle之后，项目的整体结构将会变得更加扁平化，新的项目结构将会如下所示：

```
$ tree
.
├── Contact.groovy
├── ContactController.groovy
├── ContactRepository.groovy
├── schema.sql
├── static
│   └── style.css
└── templates
    └── home.html
```

schema.sql、style.css和home.html将会保持原样，但是需要将Java类转换为Groovy。我们先从使用Groovy编写Web层开始。

21.3.1 编写Groovy控制器

如前所述，Groovy不像Java那样有很多的繁文缛节。这意味着我们在编写Groovy代码的时候，可以省略如下的内容：

- 分号；
- 像public和private这样的修饰符；
- 属性的Setter和Getter方法；
- 方法返回值的return关键字。

借助Groovy更加灵活的语法（以及Spring Boot的魔力），我们可以使用Groovy重写ContactController类，如程序清单21.8所示。

程序清单**21.8** 使用**Groovy**编写的**ContactController**要比使用**Java**更简单

```
@Grab("thymeleaf-spring4")          <─ 获得 Thymeleaf 依赖

@Controller
@RequestMapping("/")
class ContactController {

  @Autowired
  ContactRepository contactRepo                  <─ 注入 ContactRepository

  @RequestMapping(method=RequestMethod.GET)      <─ 处理对“/”的 GET 请求
   String home(Map<String,Object> model) {
    List<Contact> contacts = contactRepo.findAll()
    model.putAll([contacts: contacts])
    "home"
  }

  @RequestMapping(method=RequestMethod.POST)     <─ 处理对“/”的 POST 请求
   String submit(Contact contact) {
    contactRepo.save(contact)
    "redirect:/"
  }
}
```

我们可以看到，这个版本的ContactController要比对应的Java版本更加简洁。排除掉Groovy不需要的内容后，ContactController更加简短也更易于阅读。

程序清单21.8还移除了一些内容，你可能也发现了，这里没有import代码行，在Java代码中这是很常见的。Groovy默认会导入一些包和类，包括：

- java.io.*
- java.lang.*
- java.math.BigDecimal
- java.math.BigInteger
- java.net.*
- java.util.*
- groovy.lang.*
- groovy.util.*

因为有了这些默认的导入，所以ContactController就不需要导入List类了。这个类位于java.util包中，包含在默认的导入里面。

但是，像@Controller、@RequestMapping、@Autowired以及@RequestMethod这样的Spring类型该怎么处理呢？它们没有位于默认的导入中，我们该如何省略import代码行呢？

稍后，当我们运行应用的时候，Spring Boot CLI将会试图使用Groovy编译器编译这些Groovy类。因为这些类型没有导入进来，所以将会导

致编译失败。

但是，Spring Boot CLI却不会就这样轻易放弃，在这里CLI将自动配置达到了一个新高度。CLI将会识别出失败是因为缺少Spring类型，它会采取两个步骤来修正这个问题。首先会获取Spring Boot Web Starter依赖并将其依赖的其他内容都添加到类路径下（这样会下载并添加JAR到类路径下）。然后，它会将必要的包添加到Groovy编译器的默认导入列表中，然后重新尝试编译代码。

CLI这种自动添加依赖/自动导入的结果就是我们的控制器类不需要任何的import语句了，并且我们没有必要再手动或者通过Maven、Gradle来解析Spring库。Spring Boot CLI将会为我们完成所有的事情。

现在，让我们后退一步，考虑一下这里都发生了什么。通过在代码中使用Spring MVC类型，如`@Controller`或`@RequestMapping`，CLI将会自动解析Spring Boot Web Starter依赖。将Web Starter的依赖传递添加到类路径之后，Spring Boot的自动配置将会发挥作用，它会为我们自动配置Spring MVC功能所需的bean。不过，在这里我们需要做的仅仅是使用这些类型，Spring Boot将会处理所有的事情。

当然，CLI的功能也会有一些限制。尽管它知道如何解析众多的Spring依赖，并且能够自动将很多Spring类型（以及很多其他的库）添加到导入中，但是它不能自动解析和导入所有的功能。例如，使用Thymeleaf模板是一个可替换的方案，所以要在代码中通过`@Grab`显示声明。

还要注意，很多的依赖都没有必要指定group ID和版本号。Spring Boot将会在解析`@Grab`依赖的时候参与进来，将缺失的group ID和版本号添加上。

借助`@Grab`注解，我们声明了要使用Thymeleaf，这会触发自动配置功能，将会自动配置在Spring MVC中支持Thymeleaf模板所需的bean。

尽管Contact类与Spring Boot没有太大关系，但为了样例的完整性，我还是将它的Groovy代码展现在了下面：

```
class Contact {
  long id
  String firstName
  String lastName
  String phoneNumber
  String emailAddress
}
```

可以看到，Contact也更加简洁，没有分号、存取器方法以及像public和private这样的修饰符。这完全归功于Groovy简单的语法，其实Spring Boot并没有参与简化Contact类。

接下来，我们看一下如何借助Spring Boot CLI和Groovy来简化Repository类。

21.3.2 使用Groovy Repository实现数据持久化

ContactController中所用到的Groovy和Spring Boot CLI技巧都可以应用到ContactRepository中。如下的程序清单展现了Groovy版本的ContactRepository。

程序清单**21.9** 使用**Groovy**编写时，**ContactRepository**会更加简洁

```
@Grab("h2")                                    <-- 获取 H2
                                                   数据库的依赖
import java.sql.ResultSet

class ContactRepository {

  @Autowired
  JdbcTemplate jdbc                    <-- 注入 JdbcTemplate

  List<Contact> findAll() {                    <-- 查询联系人
     jdbc.query(
      "select id, firstName, lastName, phoneNumber, emailAddress " +
      "from contacts order by lastName",
      new RowMapper<Contact>() {
        Contact mapRow(ResultSet rs, int rowNum) {
          new Contact(id: rs.getLong(1), firstName: rs.getString(2),
              lastName: rs.getString(3), phoneNumber: rs.getString(4),
              emailAddress: rs.getString(5))
        }
      })
  }

  void save(Contact contact) {                 <-- 保存联系人
     jdbc.update(
      "insert into contacts " +
        "(firstName, lastName, phoneNumber, emailAddress) " +
        "values (?, ?, ?, ?)",
      contact.firstName, contact.lastName,
      contact.phoneNumber, contact.emailAddress)
  }
}
```

除了Groovy在语法方面带来的明显改善，这个新版的ContactRepository类使用了Spring Boot CLI自动导入JdbcTemplate和RowMapper。除此之外，当CLI发现我们使用这些类型的时候，将会自动解析JDBC Starter依赖。

只有两件事情是CLI的自动导入和自动解析无法帮助我们的。可以看到，我们依然需要导入ResultSet。另外，Spring Boot无法知道我

们使用哪种数据库，因此必须要使用`@Grab`注解添加H2数据库。

我们已经将所有Java类转换成了Groovy并在这个过程中发挥了Spring Boot的魔力。现在，我们可以运行应用了。

21.3.3 运行Spring Boot CLI

在编译完Java应用之后，有两种方法来运行它。我们可以按照可执行JAR或WAR文件的形式在命令行运行，也可以将WAR文件部署到Servlet容器中运行。Spring Boot CLI提供了第三种可选方案。

从名字应该也能猜得出来，通过Spring Boot CLI运行应用需要使用命令行。但是，借助CLI，我们不需要首先将应用构建为JAR或WAR文件。运行应用的时候，我们可以直接将Groovy源码传给CLI。

安装CLI

为了使用Spring Boot CLI，我们需要安装它。有多种方案可供选择，包括：

- Groovy环境管理器（Groovy Environment Manager，GVM）；
- Homebrew；
- 手动安装。

如果使用GVM安装CLI的话，输入以下命令：

```
$ gvm install springboot
```

你如果使用OS X的话，我们可以使用Homebrew来安装Spring Boot CLI：

```
$ brew tap pivotal/tap
$ brew install springboot
```

如果你愿意手动安装Spring Boot的话，那么可以下载并按照该站点http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/的指南进行安装。

CLI安装完成之后，可以使用如下的命令检查安装情况以及当前使用的是哪个版本：

```
$ spring --version
```

假设安装没有问题的话，那就可以运行Contacts应用了。

使用**CLI**运行**Contacts**应用

要使用Spring Boot CLI运行应用的话，我们需要在命令行输入spring run，然后后面再加上要通过CLI运行的一个或多个Groovy文件。例如，如果应用只有一个Groovy文件的话，那么可以这样运行：

```
$ spring run Hello.groovy
```

这样就会通过CLI运行一个名为Hello.groovy的Groovy类。

如果你的应用有多个Groovy类文件的话，那么可以通过通配符来运行，如下所示；

```
$ spring run *.groovy
```

或者，如果这些Groovy类文件位于同一个或多个子目录下，那么我们可以使用Ant风格的通配符递归查找Groovy类：

```
$ spring run **/*.groovy
```

因为Contacts应用有三个需要读取的Groovy类，而且它们都位于项目的根目录下，所以上述的后两种方案都是可行的。在运行应用之后，我们就能够在浏览器中访问http://localhost:8080，并且能够在浏览器中看到与之前相同的Contacts应用。

到此为止，我们以两种方式编写了Spring Boot应用：一种使用Java，另一种使用Groovy。在这两种情况中，Spring Boot在最小化模板配置以及构建依赖方面都发挥了很大的作用。Spring Boot还有另外一项功能。让我们看一下如何借助Spring Boot Actuator为Web应用引入管理端点。

21.4 通过Actuator获取了解应用内部状况

Spring Boot Actuator所完成的主要功能就是为基于Spring Boot的应用添加多个有用的管理端点。这些端点包括以下几个内容。

- `GET /autoconfig`：描述了Spring Boot在使用自动配置的时候，所做出的决策；
- `GET /beans`：列出运行应用所配置的bean；

- GET /configprops：列出应用中能够用来配置bean的所有属性及其当前的值；
- GET /dump：列出应用的线程，包括每个线程的栈跟踪信息；
- GET /env：列出应用上下文中所有可用的环境和系统属性变量；
- GET /env/{name}：展现某个特定环境变量和属性变量的值；
- GET /health：展现当前应用的健康状况；
- GET /info：展现应用特定的信息；
- GET /metrics：列出应用相关的指标，包括请求特定端点的运行次数；
- GET /metrics/{name}：展现应用特定指标项的指标状况；
- POST /shutdown：强制关闭应用；
- GET /trace：列出应用最近请求相关的元数据，包括请求和响应头。

为了启用Actuator，我们只需将Actuator Starter依赖添加到项目中即可。如果你使用Groovy编写应用并通过Spring Boot CLI来运行，那么可以通过@Grab注解来添加Actuator Starter，如下所示：

```
@Grab("spring-boot-starter-actuator")
```

如果使用Gradle构建Java应用的话，那么在build.gradle的dependencies代码块中需要添加如下的依赖：

```
compile("org.springframework.boot:spring-boot-starter-actuator")
```

或者，在项目的Maven pom.xml文件中，我们可以添加如下的<dependency>：

```
<dependency>
  <groupId> org.springframework.boot</groupId>
  <artifactId>spring-boot-actuator</carlsbad>
</dependency>
```

添加完Spring Boot Actuator之后，我们可以重新构建并启动应用，然后打开浏览器访问以上所述的端点来获取更多信息。例如，如果想要查看Spring应用上下文中所有的bean，那么可以访问http://localhost:8080/beans。如果使用curl命令行工具的话，所得到的结果将会如下所示（为了便于阅读，进行了格式化和删减）：

```
$ curl http://localhost:8080/beans
[
  {
    "beans": [
      {
        "bean": "contactController",
        "dependencies": [
          "contactRepository"
        ],
        "resource": "null",
        "scope": "singleton",
        "type": "ContactController"
      },
      {
        "bean": "contactRepository",
        "dependencies": [
          "jdbcTemplate"
        ],
        "resource": "null",
        "scope": "singleton",
        "type": "ContactRepository"
      },

      ...

      {
        "bean": "jdbcTemplate",
        "dependencies": [],
        "resource": "class path resource [...]",
        "scope": "singleton",
        "type": "org.springframework.jdbc.core.JdbcTemplate"
      },

      ...
    ]
  }
]
```

从这里，我们可以看到有一个ID为`contactController`的bean，它依赖于名为`contactRepository`的bean，而`contactRepository`又依赖于`jdbcTemplate`bean。

因为我对输出进行了删减，所以有很多的bean没有展现出来，它们都包含在“/beans”端点所产生的JSON中。对于自动装配和自动配置所形成的神秘结果，这里提供了一种了解内部实现的手段。

另外一个端点也能帮助我们了解Spring Boot自动配置的内部情况，这就是“/autoconfig”。这个端点所返回的JSON描述了Spring Boot在自动配置bean的时候所做出的决策。例如，当针对Contacts应用调用“/autoconfig”端点时，如下展现了删减后（并进行了格式化）的JSON结果：

```
$ curl http://localhost:8080/autoconfig
{
"negativeMatches": {
  "AopAutoConfiguration": [
    {
      "condition": "OnClassCondition",
      "message": "required @ConditionalOnClass classes not found:
          org.aspectj.lang.annotation.Aspect,
          org.aspectj.lang.reflect.Advice"
    }
  ],
  "BatchAutoConfiguration": [
    {
      "condition": "OnClassCondition",
      "message": "required @ConditionalOnClass classes not found:
          org.springframework.batch.core.launch.JobLauncher"
    }
  ],

  ...

},
"positiveMatches": {
  "ThymeleafAutoConfiguration": [
    {
      "condition": "OnClassCondition",
      "message": "@ConditionalOnClass classes found:
          org.thymeleaf.spring4.SpringTemplateEngine"
    }
  ],
  "ThymeleafAutoConfiguration.DefaultTemplateResolverConfiguration":[
    {
      "condition": "OnBeanCondition",
      "message": "@ConditionalOnMissingBean
          (names: defaultTemplateResolver; SearchStrategy: all)
          found no beans"
    }
  ],
  "ThymeleafAutoConfiguration.ThymeleafDefaultConfiguration": [
    {
      "condition": "OnBeanCondition",
      "message": "@ConditionalOnMissingBean (types:
          org.thymeleaf.spring4.SpringTemplateEngine;
          SearchStrategy: all) found no beans"
    }
  ],
  "ThymeleafAutoConfiguration.ThymeleafViewResolverConfiguration": [
    {
      "condition": "OnClassCondition",
      "message": "@ConditionalOnClass classes found:
          javax.servlet.Servlet"
    }
  ],
  "ThymeleafAutoConfiguration.ThymeleafViewResolverConfiguration
          #thymeleafViewResolver": [
    {
      "condition": "OnBeanCondition",
      "message": "@ConditionalOnMissingBean (names:
          thymeleafViewResolver; SearchStrategy: all)
          found no beans"
    }
  ],

  ...
}
}
```

我们可以看到，这个报告包含了两部分：一部分是没有匹配上的（negative matches），另一部分是匹配上的（positive matches）。在没有匹配的部分中，表明没有使用AOP和自动配置，因为在类路径中没有找到所需的类。在匹配上的部分中，我们可以看到，因为在类路径下找到了`SpringTemplateEngine`，Thymeleaf自动配置将会发挥作用。同时还可以看到，除非明确声明了默认的模板解析器、视图解析器以及模板bean否则的话，这些bean会进行自动配置。另外，只有在类路径中能够找到`Servlet`类，才会自动配置默认的视图解析器。

“/beans”和“/autoconfig”端点只是Spring Boot Actuator所提供的观察应用内部状况的两个样例。在本章中，我们没有足够的篇幅详细讨论每个端点，但是我建议你自行尝试这些端点，以便掌握Actuator都提供了哪些功能来帮助我们了解应用的内部状况。

21.5　小结

Spring Boot是Spring家族中一个令人兴奋的新项目。Spring致力于简化Java开发，而Spring Boot致力于让Spring本身更加简单。

Spring Boot用了两个技巧来消除Spring项目中的样板式配置：Spring Boot Starter和自动配置。

一个简单的Spring Boot Starter依赖能够替换掉Maven或Gradle构建中多个通用的依赖。例如，在项目中添加Spring Boot Web依赖后，将会引入Spring Web和Spring MVC模块，以及Jackson 2数据绑定模块。

自动配置充分利用了Spring 4.0的条件化配置特性，能够自动配置特定的Spring bean，用来启用某项特性。例如，Spring Boot能够在应用的类路径中探测到Thymeleaf，然后自动将Thymeleaf模板配置为Spring MVC视图的bean。

Spring Boot的命令行接口（command-line interface，CLI）使用Groovy进一步简化了Spring项目。通过在Groovy代码中简单地引用Spring组件，CLI就能自动添加所需的Starter依赖（而这又会触发自动配置）。除此之外，通过Spring Boot CLI运行时，很多的Spring类型都不需要在Groovy代码中显式使用import语句导入。

最后，Spring Boot Actuator为基于Spring Boot开发的Web应用提供了一些通用的管理特性，包括查看线程dump、Web请求历史以及Spring应

用上下文中的bean。

在读完本章之后，你可能会想为什么要将像Spring Boot这样有用的话题放到书的结尾呢。你甚至可能会想，如果我早一点介绍Spring Boot的话，那么很多之前所学的内容将会更加简单。确实，Spring Boot在Spring之上提供了很有意思的编程模型，一旦用上它之后，很难想象如果没有它的话，该如何编写Spring应用。

我可以说之所以将Spring Boot留在最后，是因为想让你对Spring有更深入的理解（反正对你有好处就是了）。尽管可以这么讲，但真正的原因是Spring Boot推出的时候，本书的大部分内容已经写完了。所以我只能将其放到一个不影响整本书的地方：也就是结尾。

谁知道呢？也许在本书的下一版中，从一开始我就会介绍Spring Boot。

看完了

如果您对本书内容有疑问，可发邮件至contact@epubit.com.cn，会有编辑或作译者协助答疑。也可访问异步社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@epubit.com.cn。

在这里可以找到我们：

- 微博：@人邮异步社区
- QQ群：368449889

目录

版权信息 11

作者简介 12

版权声明 13

内容提要 14

关于本书 15

路线图 15

代码规范与下载 17

作者在线 17

封面插图简介 18

前言 19

译者序 21

致谢 23

第1部分 Spring的核心 25

第1章 Spring之旅 26

1.1 简化Java开发 27

1.1.1 激发POJO的潜能 27

1.1.2 依赖注入 28

1.1.3 应用切面 35

1.1.4 使用模板消除样板式代码 40

1.2 容纳你的Bean 43

1.2.1 使用应用上下文 44

1.2.2 bean的生命周期 45

1.3 俯瞰Spring风景线 47

1.3.1 Spring模块 47

1.3.2 Spring Portfolio 51

1.4 Spring的新功能 54

1.4.1 Spring 3.1新特性 55

1.4.2 Spring 3.2新特性 56

1.4.3 Spring 4.0新特性 57

1.5 小结 58

第2章 装配Bean 60
2.1 Spring配置的可选方案 61
2.2 自动化装配bean 62
2.2.1 创建可被发现的bean 62
2.2.2 为组件扫描的bean命名 66
2.2.3 设置组件扫描的基础包 67
2.2.4 通过为bean添加注解实现自动装配 68
2.2.5 验证自动装配 70
2.3 通过Java代码装配bean 72
2.3.1 创建配置类 72
2.3.2 声明简单的bean 73
2.3.3 借助JavaConfig实现注入 74
2.4 通过XML装配bean 76
2.4.1 创建XML配置规范 77
2.4.2 声明一个简单的<bean> 78
2.4.3 借助构造器注入初始化bean 79
2.4.4 设置属性 86
2.5 导入和混合配置 92
2.5.1 在JavaConfig中引用XML配置 92
2.5.2 在XML配置中引用JavaConfig 94
2.6 小结 96
第3章 高级装配 98
3.1 环境与profile 98
3.1.1 配置profile bean 100
3.1.2 激活profile 104
3.2 条件化的bean 107
3.3 处理自动装配的歧义性 111
3.3.1 标示首选的bean 112
3.3.2 限定自动装配的bean 113
3.4 bean的作用域 117
3.4.1 使用会话和请求作用域 119
3.4.2 在XML中声明作用域代理 121
3.5 运行时值注入 122
3.5.1 注入外部的值 123

3.5.2 使用Spring表达式语言进行装配 127
3.6 小结 135
第4章 面向切面的Spring 137
4.1 什么是面向切面编程 138
4.1.1 定义AOP术语 139
4.1.2 Spring对AOP的支持 142
4.2 通过切点来选择连接点 144
4.2.1 编写切点 146
4.2.2 在切点中选择bean 147
4.3 使用注解创建切面 147
4.3.1 定义切面 148
4.3.2 创建环绕通知 152
4.3.3 处理通知中的参数 153
4.3.4 通过注解引入新功能 157
4.4 在XML中声明切面 160
4.4.1 声明前置和后置通知 162
4.4.2 声明环绕通知 164
4.4.3 为通知传递参数 165
4.4.4 通过切面引入新的功能 167
4.5 注入AspectJ切面 168
4.6 小结 171
第2部分 Web中的Spring 173
第5章 构建Spring Web应用程序 174
5.1 Spring MVC起步 174
5.1.1 跟踪Spring MVC的请求 175
5.1.2 搭建Spring MVC 176
5.1.3 Spittr应用简介 182
5.2 编写基本的控制器 183
5.2.1 测试控制器 185
5.2.2 定义类级别的请求处理 186
5.2.3 传递模型数据到视图中 187
5.3 接受请求的输入 193
5.3.1 处理查询参数 194
5.3.2 通过路径参数接受输入 196

5.4 处理表单 199
5.4.1 编写处理表单的控制器 202
5.4.2 校验表单 205
5.5 小结 208
第6章 渲染Web视图 210
6.1 理解视图解析 210
6.2 创建JSP视图 213
6.2.1 配置适用于JSP的视图解析器 214
6.2.2 使用Spring的JSP库 216
6.3 使用Apache Tiles视图定义布局 231
6.3.1 配置Tiles视图解析器 232
6.4 使用Thymeleaf 237
6.4.1 配置Thymeleaf视图解析器 238
6.4.2 定义Thymeleaf模板 240
6.5 小结 244
第7章 Spring MVC的高级技术 246
7.1 Spring MVC配置的替代方案 246
7.1.1 自定义DispatcherServlet配置 247
7.1.2 添加其他的Servlet和Filter 248
7.1.3 在web.xml中声明DispatcherServlet 250
7.2 处理multipart形式的数据 253
7.2.1 配置multipart解析器 255
7.2.2 处理multipart请求 259
7.3 处理异常 263
7.3.1 将异常映射为HTTP状态码 263
7.3.2 编写异常处理的方法 266
7.4 为控制器添加通知 268
7.5 跨重定向请求传递数据 269
7.5.1 通过URL模板进行重定向 270
7.5.2 使用flash属性 271
7.6 小结 273
第8章 使用Spring Web Flow 275
8.1 在Spring中配置Web Flow 276
8.1.1 装配流程执行器 276

8.1.2 配置流程注册表 276
8.1.3 处理流程请求 278
8.2 流程的组件 278
8.2.1 状态 279
8.2.2 转移 283
8.2.3 流程数据 284
8.3 组合起来：披萨流程 286
8.3.1 定义基本流程 287
8.3.2 收集顾客信息 291
8.3.3 构建订单 296
8.3.4 支付 299
8.4 保护Web流程 302
8.5 小结 302
第9章 保护Web应用 304
9.1 Spring Security简介 305
9.1.1 理解Spring Security的模块 305
9.1.2 过滤Web请求 307
9.1.3 编写简单的安全性配置 308
9.2 选择查询用户详细信息的服务 311
9.2.1 使用基于内存的用户存储 311
9.2.2 基于数据库表进行认证 314
9.2.3 基于LDAP进行认证 316
9.2.4 配置自定义的用户服务 320
9.3 拦截请求 322
9.3.1 使用Spring表达式进行安全保护 325
9.3.2 强制通道的安全性 327
9.3.3 防止跨站请求伪造 329
9.4 认证用户 330
9.4.1 添加自定义的登录页 331
9.4.2 启用HTTP Basic认证 333
9.4.3 启用Remember-me功能 334
9.4.4 退出 335
9.5 保护视图 336
9.5.1 使用Spring Security的JSP标签库 336

9.5.2 使用Thymeleaf的Spring Security方言 340
9.6 小结 342
第3部分 后端中的Spring 343
第10章 通过Spring和JDBC征服数据库 344
10.1 Spring的数据访问哲学 344
10.1.1 了解Spring的数据访问异常体系 346
10.1.2 数据访问模板化 348
10.2 配置数据源 351
10.2.1 使用JNDI数据源 351
10.2.2 使用数据源连接池 352
10.2.3 基于JDBC驱动的数据源 354
10.2.4 使用嵌入式的数据源 355
10.2.5 使用profile选择数据源 357
10.3 在Spring中使用JDBC 359
10.3.1 应对失控的JDBC代码 360
10.3.2 使用JDBC模板 363
10.4 小结 368
第11章 使用对象-关系映射持久化数据 369
11.1 在Spring中集成Hibernate 370
11.1.1 声明Hibernate的Session工厂 371
11.1.2 构建不依赖于Spring的Hibernate代码 373
11.2 Spring与Java持久化API 375
11.2.1 配置实体管理器工厂 375
11.2.2 编写基于JPA的Repository 382
11.3 借助Spring Data实现自动化的JPA Repository 385
11.3.1 定义查询方法 387
11.3.2 声明自定义查询 391
11.3.3 混合自定义的功能 392
11.4 小结 394
第12章 使用NoSQL数据库 396
12.1 使用MongoDB持久化文档数据 396
12.1.1 启用MongoDB 398
12.1.2 为模型添加注解，实现MongoDB持久化 401

12.1.3 使用MongoTemplate访问MongoDB 405
12.1.4 编写MongoDB Repository 407
12.2 使用Neo4j操作图数据 413
12.2.1 配置Spring Data Neo4j 413
12.2.2 使用注解标注图实体 416
12.2.3 使用Neo4jTemplate 420
12.2.4 创建自动化的Neo4j Repository 422
12.3 使用Redis操作key-value数据 428
12.3.1 连接到Redis 428
12.3.2 使用RedisTemplate 430
12.3.3 使用key和value的序列化器 434
12.4 小结 436
第13章 缓存数据 437
13.1 启用对缓存的支持 438
13.1.1 配置缓存管理器 439
13.2 为方法添加注解以支持缓存 444
13.2.1 填充缓存 445
13.2.2 移除缓存条目 450
13.3 使用XML声明缓存 451
13.4 小结 456
第14章 保护方法应用 457
14.1 使用注解保护方法 457
14.1.1 使用@Secured注解限制方法调用 458
14.1.2 在Spring Security中使用JSR-250的@RolesAllowed注解 460
14.2 使用表达式实现方法级别的安全性 461
14.2.1 表述方法访问规则 462
14.2.2 过滤方法的输入和输出 464
14.3 小结 469
第4部分 Spring集成 471
第15章 使用远程服务 472
15.1 Spring远程调用概览 473
15.2 使用RMI 475
15.2.1 导出RMI服务 476

15.2.2 装配RMI服务 479
15.3 使用Hessian和Burlap发布远程服务 482
15.3.1 使用Hessian和Burlap导出bean的功能 482
15.3.2 访问Hessian/Burlap服务 486
15.4 使用Spring的HttpInvoker 488
15.4.1 将bean导出为HTTP服务 488
15.4.2 通过HTTP访问服务 490
15.5 发布和使用Web服务 491
15.5.1 创建基于Spring的JAX-WS端点 491
15.5.2 在客户端代理JAX-WS服务 495
15.6 小结 497
第16章 使用Spring MVC创建REST API 498
16.1 了解REST 498
16.1.1 REST的基础知识 499
16.1.2 Spring是如何支持REST的 500
16.2 创建第一个REST端点 501
16.2.1 协商资源表述 503
16.2.2 使用HTTP信息转换器 510
16.3 提供资源之外的其他内容 516
16.3.1 发送错误信息到客户端 517
16.3.2 在响应中设置头部信息 522
16.4 编写REST客户端 524
16.4.1 了解RestTemplate的操作 525
16.4.2 GET资源 527
16.4.3 检索资源 528
16.4.4 抽取响应的元数据 529
16.4.5 PUT资源 530
16.4.6 DELETE资源 532
16.4.7 POST资源数据 533
16.4.8 在POST请求中获取响应对象 533
16.4.9 在POST请求后获取资源位置 535
16.4.10 交换资源 535
16.5 小结 538
第17章 Spring消息 539

17.1 异步消息简介 540
17.1.1 发送消息 541
17.1.2 评估异步消息的优点 544
17.2 使用JMS发送消息 545
17.2.1 在Spring中搭建消息代理 546
17.2.2 使用Spring的JMS模板 548
17.2.3 创建消息驱动的POJO 558
17.2.4 使用基于消息的RPC 562
17.3 使用AMQP实现消息功能 565
17.3.1 AMQP简介 565
17.3.2 配置Spring支持AMQP消息 567
17.3.3 使用RabbitTemplate发送消息 570
17.3.4 接收AMQP消息 573
17.4 小结 575
第18章 使用WebSocket和STOMP实现消息功能 577
18.1 使用Spring的低层级WebSocket API 577
18.2 应对不支持WebSocket的场景 583
18.3 使用STOMP消息 586
18.3.1 启用STOMP消息功能 588
18.3.2 处理来自客户端的STOMP消息 591
18.3.3 发送消息到客户端 595
18.4 为目标用户发送消息 599
18.4.1 在控制器中处理用户的消息 600
18.4.2 为指定用户发送消息 602
18.5 处理消息异常 603
18.6 小结 604
第19章 使用Spring发送Email 606
19.1 配置Spring发送邮件 606
19.1.1 配置邮件发送器 607
19.1.2 装配和使用邮件发送器 608
19.2 构建丰富内容的Email消息 610
19.2.1 添加附件 610
19.2.2 发送富文本内容的Email 611
19.3 使用模板生成Email 613

19.3.1 使用Velocity构建Email消息 613
19.3.2 使用Thymeleaf构建Email消息 615
19.4 小结 618
第20章 使用JMX管理Spring Bean 619
20.1 将Spring bean导出为MBean 620
20.1.1 通过名称暴露方法 623
20.1.2 使用接口定义MBean的操作和属性 625
20.1.3 使用注解驱动的MBean 627
20.1.4 处理MBean冲突 628
20.2 远程MBean 629
20.2.1 暴露远程MBean 630
20.2.2 访问远程MBean 631
20.2.3 代理MBean 633
20.3 处理通知 634
20.3.1 监听通知 636
20.4 小结 637
第21章 借助Spring Boot简化Spring开发 638
21.1 Spring Boot简介 639
21.1.1 添加Starter依赖 639
21.1.2 自动配置 644
21.1.3 Spring Boot CLI 644
21.1.4 Actuator 645
21.2 使用Spring Boot构建应用 646
21.2.1 处理请求 648
21.2.2 创建视图 650
21.2.3 添加静态内容 652
21.2.4 持久化数据 654
21.2.5 尝试运行 656
21.3 组合使用Groovy与Spring Boot CLI 659
21.3.1 编写Groovy控制器 660
21.3.2 使用Groovy Repository实现数据持久化 663
21.3.3 运行Spring Boot CLI 664
21.4 通过Actuator获取了解应用内部状况 665
21.5 小结 669

Spring
实战（第4版）

本书是经典的 Spring 框架的实用指南。第 4 版涵盖了 Spring 的核心知识以及 Spring MVC、Security、Web Flow 等最新功能。在学习如何构建简单和实际的 JEE 应用过程中，你将会看到简短的代码片段和不断完善的样例。本书作者通过简短有趣的企业级样例，让读者掌握真正需要的特性和技术。

Spring 框架最初设计于 2003 年，作为 J2EE 开发的轻量级解决方案，目前已经成为构建企业级应用的标准选择和 Java 开发人员的必备工具。Spring 4 是最新的主版本，提供了对 Java 8 的集成和重要的升级，包括 IoC 容器的新注解、Spring 表达式语言的增强以及 REST 所需的支持。

本书涵盖的内容：

- Spring 4 的升级
- 用于 NoSQL 的 Spring Data
- 通过注解和 profile 定义简化配置
- 使用 RESTful 资源

不管你是刚刚接触 Spring 还是想了解一些新特性，本书都是掌握 Spring 的绝佳选择。

Craig Walls 是 Pivotal 的软件开发者。他是畅销书作者，并且经常在用户群组和技术大会做演讲。Craig 住在 Texas 的 Cross Roads。

这是最好的一本 Spring 书籍。第 4 版进行了全面的升级和修订。

——Gregor Zurowski，Sotheby’s

经典好书，重新升级并且具有权威性。

——Mario Arias，Cake Solutions Ltd

内容丰富，精确且具有洞见性。

——Jeelani Shaik，D3Banking.com

十年的时间过去了，它依然是最清晰、最综合地介绍 Spring 平台核心理念的权威指南。

——James Wright，Sword-Apak

MANNING

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机／程序设计
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-41730-5

ISBN 978-7-115-41730-5
定价:89.00 元